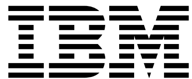
XL Fortran for AIX

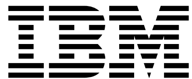


Language Reference

Version 7, Release 1

SC09-2867-00

XL Fortran for AIX



Language Reference

Version 7, Release 1

SC09-2867-00

Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 747.

First Edition (February 2000)

This edition applies to Version 7, Release 1 of IBM XL Fortran for AIX (5765–E02) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

IBM welcomes your comments. You can send your comments by any of the following methods:

- Electronically to the network ID listed below. Be sure to include your entire network address if you wish a reply.
  - Internet: torrcf@ca.ibm.com
  - IBMLink: toribm(torrcf)
- By FAX, use the following number:
  - United States and Canada: 416-448-6161
  - Other countries: (+1)-416-448-6161
- By mail to the following address:

IBM Canada Ltd. Laboratory
Information Development
2G/KB7/1150/TOR
1150 Eglinton Avenue East
North York, Ontario, Canada. M3C 1H7

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1990, 2000. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Summary of Changes for XL Fortran . . . xi
OpenMP Fortran API, Version 1.0 Features . . xi
Components of the XL Fortran Language . . xi
Highlighting Conventions xii

How to Use This Book. xiii
Highlighting Conventions xiv
How to Read Syntax Diagrams xiv
Syntax Diagrams xiv
Example of a Syntax Diagram. xvi
Notes on the Examples in This Book . . . xvi
Related Documentation xvi
Standards Documents xvii

Part 1. Introduction. 1

Chapter 1. Introduction to XL Fortran for AIX 3
XL Fortran (XLF). 3
What is OpenMP? 4
Compiler Directives 4
Library Routines 4
Environment Variables 5
What is Fortran 95? 5
FORALL 5
PURE 5
ELEMENTAL 5
Initialization 6
Specification Functions. 6
Deleted Features 6
What is Fortran 90? 6
Fortran 90 Free Source Form 6
Parameterized Data Types. 7
Derived Types. 7
Array Enhancements 7
Pointers 7
Dynamic Behavior 7
Control Construct Enhancements 8
Procedure Enhancements 8
Modules 8
New Intrinsic Procedures 9
Valid and Invalid XL Fortran Programs . . . 9

Part 2. Concepts and Elements . . 11

Chapter 2. The Language Elements . . . 13
Characters 13
Names 14
Statements 15
Statement Keywords 15
Statement Labels 15
Lines and Source Formats 16
Fixed Source Form 17
Free Source Form 19
IBM Free Source Form 21
Conditional Compilation 22
Order of Statements and Execution Sequence 25

Chapter 3. Data Types and Data Objects 27
Data Types 27
Type Parameters and Specifiers 27
Data Objects 28
Constants 28
Automatic Objects 29
Intrinsic Types 29
Integer 29
Real. 30
Complex 33
Logical. 34
Character 35
BYTE 38
Derived Types 39
Determining Type for Derived Types. . . 42
Typeless Literal Constants 46
Hexadecimal Constants 46
Octal Constants 47
Binary Constants 48
Hollerith Constants 48
Using Typeless Constants 49
How Type Is Determined 52
Definition Status of Variables 53
Events Causing Definition 53
Events Causing Undefinition 55
Allocation Status 58
Storage Classes for Variables 59
Fundamental Storage Classes 59
Secondary Storage Classes 59
Storage Class Assignment 60

Chapter 4. Array Concepts 63

© Copyright IBM Corp. 1990, 2000

Arrays 63
Bounds of a Dimension 63
Extent of a Dimension 64
Rank, Shape, and Size of an Array . . . 64
Array Declarators 65
Explicit-Shape Arrays 66
Examples of Explicit-Shape Arrays . . . 66
Automatic Arrays 66
Adjustable Arrays 67
Pointee Arrays 67
Assumed-Shape Arrays 68
Examples of Assumed-Shape Arrays . . . 69
Deferred-Shape Arrays 69
Allocatable Arrays 70
Array Pointers 71
Assumed-Size Arrays 71
Examples of Assumed-Size Arrays . . . 72
Array Elements 73
Notes 74
Array Element Order 74
Array Sections 74
Subscript Triplets 76
Vector Subscripts 78
Array Sections and Substring Ranges . . 78
Array Sections and Structure Components 79
Rank and Shape of Array Sections . . . 80
Array Constructors. 81
Implied-DO List for an Array Constructor 81
Expressions Involving Arrays 82

Chapter 5. Expressions and Assignment 85
Introduction to Expressions and Assignment 85
Primary 86
Constant Expressions 87
Examples of Constant Expressions . . . 87
Initialization Expressions. 87
Examples of Initialization Expressions . . 88
Specification Expressions. 88
Examples of Specification Expressions . . 90
Operators and Expressions 90
General 90
Arithmetic 91
Character 93
Relational 94
Logical. 96
Primary 99
Extended Intrinsic and Defined Operations . 99
How Expressions Are Evaluated 100
Precedence of Operators 100
Using BYTE Data Objects 103
Intrinsic Assignment 103
Arithmetic Conversion 105
WHERE Construct 106
Interpreting Masked Array Assignments 108
FORALL Construct 113
Interpreting the FORALL Construct . . . 115
Pointer Assignment 117
Examples of Pointer Assignment 118
Integer Pointer Assignment 118

Chapter 6. Control 121
Statement Blocks 121
IF Construct 121
Example 123
CASE Construct 123
Examples 125
DO Construct 126
The Terminal Statement. 126
DO WHILE Construct 130
Example 131
Branching 131

Chapter 7. Program Units and Procedures 133
Scope 133
The Scope of a Name 134
Association 137
Host Association 137
Use Association 139
Pointer Association 139
Integer Pointer Association. 140
Program Units, Procedures, and Subprograms 141
Internal Procedures 141
Interface Concepts 142
Interface Blocks 144
Example of an Interface. 146
Generic Interface Blocks. 147
Unambiguous Generic Procedure References 147
Extending Intrinsic Procedures with Generic Interface Blocks. 149
Defined Operators 150
Defined Assignment 151
Main Program 152
Modules 153
Example of a Module 156
Block Data Program Unit 156
Example of a Block Data Program Unit 157
Function and Subroutine Subprograms. . . 157
Procedure References 159

Intrinsic Procedures 160
Conflicts Between Intrinsic Procedure Names and Other Names 161
Arguments 161
Actual Argument Specification 161
Argument Association 163
%VAL and %REF 165
Intent of Dummy Arguments 166
Optional Dummy Arguments 167
Restrictions on Optional Dummy Arguments Not Present 168
Length of Character Arguments 168
Variables as Dummy Arguments 168
Pointers as Dummy Arguments 170
Procedures as Dummy Arguments . . . 171
Asterisks as Dummy Arguments 172
Resolution of Procedure References . . . 172
Rules for Resolving Procedure References to Names 173
Resolving Procedure References to Generic Names 174
Recursion 175
Pure Procedures 176
Examples 177
Elemental Procedures 178
Examples 180

Chapter 8. Input/Output Concepts 183
Records 183
Formatted Records 183
Unformatted Records 184
Endfile Records 184
Files 184
External Files 184
External File Access Modes: Sequential or Direct 185
Internal Files 185
Units 186
Connection of a Unit. 186
Executing Data Transfer Statements . . . 188
Executing Data Transfer Statements Asynchronously 189
Advancing and Nonadvancing Input/Output 191
File Position Before and After Data Transfer 191
Conditions and IOSTAT Values 193
End-Of-Record Conditions 193
End-Of-File Conditions 193
Error Conditions 193

Chapter 9. Input/Output Formatting . . . 201
Format-Directed Formatting 201
Data Edit Descriptors 201
Control Edit Descriptors 202
Character String Edit Descriptors . . . 203
Editing 204
Complex Editing 205
Data Edit Descriptors 205
A (Character) Editing 205
B (Binary) Editing. 206
E, D, and Q (Extended Precision) Editing 207
EN Editing 209
ES Editing 210
F (Real without Exponent) Editing . . . 211
G (General) Editing 212
I (Integer) Editing. 214
L (Logical) Editing 215
O (Octal) Editing 216
Q (Character Count) Editing 217
Z (Hexadecimal) Editing 219
Control Edit Descriptors 220
/ (Slash) Editing 220
: (Colon) Editing 221
$ (Dollar) Editing 221
Apostrophe/Double Quotation Mark Editing (Character-String Edit Descriptor). 221
BN (Blank Null) and BZ (Blank Zero) Editing 222
H Editing 223
P (Scale Factor) Editing 224
S, SP, and SS (Sign Control) Editing. . . 224
T, TL, TR, and X (Positional) Editing . . 225
Interaction between Input/Output Lists and Format Specifications 226
List-Directed Formatting 227
List-Directed Input 228
List-Directed Output 229
Namelist Formatting 231
Namelist Input Data 231
Namelist Output Data 236

Part 3. Statements and Directives 239

Chapter 10. Statements 241
Attributes 244
ALLOCATABLE 245
ALLOCATE. 246
ASSIGN 248
AUTOMATIC 250
BACKSPACE 251

BLOCK DATA 252
BYTE 253
CALL 257
CASE 258
CHARACTER 260
CLOSE 265
COMMON 267
COMPLEX 270
CONTAINS 274
CONTINUE. 275
CYCLE 276
DATA. 277
DEALLOCATE. 280
Derived Type 282
DIMENSION 283
DO 284
DO WHILE 286
DOUBLE COMPLEX. 287
DOUBLE PRECISION 290
ELSE 294
ELSE IF 294
ELSEWHERE 295
END 296
END (Construct) 298
END INTERFACE. 300
END TYPE 301
ENDFILE 302
ENTRY 303
EQUIVALENCE 306
EXIT 308
EXTERNAL. 310
FORALL 311
FORALL (Construct) 314
FORMAT 315
FUNCTION. 320
GO TO (Assigned) 324
GO TO (Computed) 325
GO TO (Unconditional) 326
IF (Arithmetic) 327
IF (Block) 328
IF (Logical) 329
IMPLICIT 329
INQUIRE 332
INTEGER 338
INTENT 342
INTERFACE 344
INTRINSIC 345
LOGICAL 347
MODULE 350
MODULE PROCEDURE 351
NAMELIST 353
NULLIFY 354
OPEN. 355
OPTIONAL 361
PARAMETER 362
PAUSE 364
POINTER (Fortran 90) 364
POINTER (integer) 366
PRINT 368
PRIVATE. 370
PROGRAM 372
PUBLIC 373
READ. 374
REAL 381
RETURN. 385
REWIND 387
SAVE 388
SELECT CASE 390
SEQUENCE. 391
Statement Function 392
STATIC 394
STOP 395
SUBROUTINE 396
TARGET 398
TYPE 399
Type Declaration 402
USE 408
VIRTUAL 410
VOLATILE 410
WAIT 412
WHERE 414
WRITE 416

Chapter 11. Directives 423
SMP and Thread-Safing Directives 423
Noncomment and Comment Form Directives 424
  Noncomment Form Directives 424
  Comment Form Directives 425
Data Scope Attribute Clauses 430
  Format 430
  Rules 432
  Examples 434
  List of Data Scope Attribute Clauses . . 436
Detailed Descriptions of Compiler Directives 445
ASSERT 445
ATOMIC. 448
BARRIER 450
CNCALL 452
CRITICAL / END CRITICAL 453
DO / END DO 455

DO SERIAL 460
EJECT 461
FLUSH 462
INCLUDE 464
INDEPENDENT 466
#line 470
MASTER / END MASTER. 472
ORDERED / END ORDERED 473
PARALLEL / END PARALLEL 476
PARALLEL DO / END PARALLEL DO . . 479
PARALLEL SECTIONS / END PARALLEL
SECTIONS 483
PERMUTATION 487
PREFETCH_BY_LOAD /
PREFETCH_FOR_LOAD /
PREFETCH_FOR_STORE 488
@PROCESS 490
SCHEDULE. 491
SECTIONS / END SECTIONS 498
SINGLE / END SINGLE 502
SOURCEFORM 506
THREADLOCAL 507
THREADPRIVATE 510
UNROLL 514

Part 4. Intrinsic and Library Procedures 517

Chapter 12. Intrinsic Procedures 519
Classes of Intrinsic Procedures 519
Inquiry Intrinsic Functions. 519
Elemental Intrinsic Procedures 519
System Inquiry Intrinsic Functions . . . 520
Transformational Intrinsic Functions . . 520
Intrinsic Subroutines. 521
Data Representation Models 521
Integer Bit Model 521
Integer Data Model 522
Real Data Model 522
Detailed Descriptions of Intrinsic Procedures 523
ABORT(). 524
ABS(A) 524
ACHAR(I) 525
ACOS(X). 525
ACOSD(X) 526
ADJUSTL(STRING) 527
ADJUSTR(STRING) 527
AIMAG(Z), IMAG(Z) 527
AINT(A, KIND) 528
ALL(MASK, DIM) 529
ALLOCATED(ARRAY) 529
ANINT(A, KIND). 530
ANY(MASK, DIM) 531
ASIN(X) 532
ASIND(X) 532
ASSOCIATED(POINTER, TARGET) 533
ATAN(X). 534
ATAND(X) 534
ATAN2(Y, X) 535
ATAN2D(Y, X) 536
BIT_SIZE(I) 537
BTEST(I, POS) 537
CEILING(A, KIND) 538
CHAR(I, KIND) 539
CMPLX(X, Y, KIND) 539
CONJG(Z) 540
COS(X) 541
COSD(X). 541
COSH(X). 542
COUNT(MASK, DIM) 542
CPU_TIME(TIME) 543
CSHIFT(ARRAY, SHIFT, DIM) 545
CVMGx(TSOURCE, FSOURCE, MASK) . . 546
DATE_AND_TIME(DATE, TIME, ZONE,
VALUES) 547
DBLE(A). 549
DCMPLX(X, Y) 550
DIGITS(X) 551
DIM(X, Y) 551
DOT_PRODUCT(VECTOR_A, VECTOR_B) 552
DPROD(X, Y) 553
EOSHIFT(ARRAY, SHIFT, BOUNDARY,
DIM) 553
EPSILON(X) 555
ERF(X) 556
ERFC(X) 556
EXP(X) 557
EXPONENT(X) 558
FLOOR(A, KIND). 558
FMADD(A, X, Y) 559
FMSUB(A, X, Y) 559
FNMADD(A, X, Y) 560
FNMSUB(A, X, Y). 560
FRE(X) 561
FRSQRTE(X) 561
FSEL(X,Y,Z) 562
FRACTION(X) 562
GAMMA(X) 563
GETENV(NAME, VALUE) 563

HFIX(A) 564
HUGE(X) 565
IACHAR(C). 565
IAND(I, J) 566
IBCLR(I, POS) 567
IBITS(I, POS, LEN) 567
IBSET(I, POS) 568
ICHAR(C) 569
IEOR(I, J) 569
ILEN(I) 570
IMAG(Z). 570
INDEX(STRING, SUBSTRING, BACK) . . . 570
INT(A, KIND) 571
IOR(I, J) 572
ISHFT(I, SHIFT) 573
ISHFTC(I, SHIFT, SIZE) 574
KIND(X) 574
LBOUND(ARRAY, DIM) 575
LEADZ(I) 576
LEN(STRING) 576
LEN_TRIM(STRING). 577
LGAMMA(X) 577
LGE(STRING_A, STRING_B) 578
LGT(STRING_A, STRING_B) 579
LLE(STRING_A, STRING_B) 579
LLT(STRING_A, STRING_B) 580
LOC(X) 581
LOG(X) 581
LOG10(X) 582
LOGICAL(L, KIND) 583
LSHIFT(I, SHIFT) 583
MATMUL(MATRIX_A, MATRIX_B, MINDIM) 584
MAX(A1, A2, A3, ...) 586
MAXEXPONENT(X) 587
MAXLOC(ARRAY, DIM, MASK) or MAXLOC(ARRAY, MASK). 588
MAXVAL(ARRAY, DIM, MASK) or MAXVAL(ARRAY, MASK) 589
MERGE(TSOURCE, FSOURCE, MASK) . . . 590
MIN(A1, A2, A3, ...) 591
MINEXPONENT(X) 592
MINLOC(ARRAY, DIM, MASK) or MINLOC(ARRAY, MASK) 593
MINVAL(ARRAY, DIM, MASK) or MINVAL(ARRAY, MASK) 594
MOD(A, P) 596
MODULO(A, P) 596
MVBITS(FROM, FROMPOS, LEN, TO, TOPOS) 597
NEAREST(X,S) 598
NINT(A, KIND) 598
NOT(I) 599
NULL(MOLD) 599
NUM_PARTHDS() 601
NUMBER_OF_PROCESSORS(DIM) 601
NUM_USRTHDS() 602
PACK(ARRAY, MASK, VECTOR) 602
PRECISION(X) 603
PRESENT(A) 604
PROCESSORS_SHAPE() 605
PRODUCT(ARRAY, DIM, MASK) or PRODUCT(ARRAY, MASK) 605
QCMPLX(X, Y) 607
QEXT(A). 607
RADIX(X) 608
RAND() 608
RANDOM_NUMBER(HARVEST) 609
RANDOM_SEED(SIZE, PUT, GET, GENERATOR) 609
RANGE(X) 611
REAL(A, KIND) 612
REPEAT(STRING, NCOPIES) 613
RESHAPE(SOURCE, SHAPE, PAD, ORDER) 613
RRSPACING(X) 614
RSHIFT(I, SHIFT) 615
SCALE(X,I) 615
SCAN(STRING, SET, BACK) 616
SELECTED_INT_KIND(R) 616
SELECTED_REAL_KIND(P, R) 617
SET_EXPONENT(X,I) 618
SHAPE(SOURCE). 618
SIGN(A, B) 619
SIGNAL(I, PROC) 620
SIN(X) 621
SIND(X) 622
SINH(X) 622
SIZE(ARRAY, DIM) 623
SPACING(X) 623
SPREAD(SOURCE, DIM, NCOPIES) . . . 624
SQRT(X) 625
SRAND(SEED). 626
SUM(ARRAY, DIM, MASK) or SUM(ARRAY, MASK) 626
SYSTEM(CMD, RESULT) 628
SYSTEM_CLOCK(COUNT, COUNT_RATE, COUNT_MAX) 629
TAN(X) 630
TAND(X) 631
TANH(X) 631

TINY(X) 632
TRANSFER(SOURCE, MOLD, SIZE) . . . 632
TRANSPOSE(MATRIX) 633
TRIM(STRING) 634
UBOUND(ARRAY, DIM) 634
UNPACK(VECTOR, MASK, FIELD). . . . 635
VERIFY(STRING, SET, BACK) 636

Chapter 13. Service and Utility Procedures 639
Efficient Floating-point Control and Inquiry Procedures 639
General Service and Utility Procedures. . . 642
List of Service and Utility Procedures . . . 644

Chapter 14. OpenMP Execution Environment Routines and Lock Routines 667

Chapter 15. Pthreads Library Module . . 677
The Pthreads Data Structures 678
Functions That Perform Operations on Thread Attribute Objects 678
Functions and Subroutines That Perform Operations on Thread 678
Functions That Perform Operations on Mutex Attribute Objects. 678
Functions That Perform Operations on Mutex Objects 678
Functions That Perform Operations on Attribute Objects of Condition Variables . 678
Functions That Perform Operations on Condition Variable Objects. 678
Functions That Perform Operations on Thread-Specific Data 679
Functions and Subroutines That Perform Operations to Control Thread Cancelability 679
Functions That Perform Operations for One-Time Initialization 679
Limitation and Caveats on the Use of the Argument arg 727

Part 5. Appendixes 729

Appendix A. Compatibility Across Standards 731
Fortran 90 compatibility. 732
Obsolescent Features. 733
Deleted Features 735

Appendix B. ASCII and EBCDIC Character Sets 737

Notices 747
Trademarks and Service Marks 750

Glossary 751

INDEX 761

Summary of Changes for XL Fortran

This section describes the differences between XL Fortran (XLF) Version 7.1. and Version 6.1. If you are familiar with previous versions of the XLF compiler, or other Fortran 95, Fortran 90 or FORTRAN 77 compilers, you will find this section useful.

OpenMP Fortran API, Version 1.0 Features

The OpenMP Fortran API provides additional features which you can use to supplement the existing FORTRAN 77, Fortran 90 and Fortran 95 language standards. See "What is OpenMP?" on page 4 for more information.

The OpenMP Architecture Review Board (ARB) is responding to questions of interpretation about aspects of the API. Some of these questions may relate to interface features that have been implemented in this version of the XL Fortran compiler. Any answers given by this committee that are related to the interface may result in changes in future releases of the XL Fortran compiler, even if these changes result in incompatibilities with previous releases of the product.

Components of the XL Fortran Language

XL Fortran Version 7.1 provides the following features:
- The full function of XL Fortran Version 6.1
- Support for the following items, which are new in Version 7.1:
  - **-qunroll** compiler option, to control whether **DO** loops can be automatically unrolled by the compiler
  - **UNROLL** directive
  - Regular expression support for **-qipa** suboptions
  - **-qhot=vector** suboption, which replaces code with calls to vector library routines
  - **-O5** compiler suboption
  - nopteovrlp suboption for the **-qalias** compiler option, to provide an alternative for optimistic pointee aliasing
  - Intrinsics for specific PowerPC operations
  - **-qsmp=rec_locks** suboption, to specify whether recursive locks are to be used to implement critical constructs
  - **PREFETCH** directives
  - **strictnmaf** suboption for the **-qfloat** compiler option

© Copyright IBM Corp. 1990, 2000

- Support for the xlf_fp_util service and utility module
- Support for the OpenMP Fortran API, Version 1.0, as understood and interpreted by IBM. The following items are new in Version 7.1:
  - **COPYIN** clause in directives
  - **ATOMIC** directive
  - **ORDERED** directive and **ORDERED** clause of the **DO** (work sharing) directive
  - **FLUSH** directive
  - **omp_destroy_lock** lock routine
  - **omp_get_dynamic** execution environment routine
  - **omp_get_max_threads** execution environment routine
  - **omp_get_nested** execution environment routine
  - **omp_get_num_procs** execution environment routine
  - **omp_init_lock** lock routine
  - **omp_in_parallel** execution environment routine
  - **omp_set_dynamic** execution environment routine
  - **omp_set_lock** lock routine
  - **omp_set_nested** execution environment routine
  - **omp_set_num_threads** execution environment routine
  - **omp_test_lock** lock routine
  - **omp_unset_lock** lock routine
  - **THREADPRIVATE** directive
  - **SINGLE/END SINGLE** directives and **SINGLE** construct
  - **SECTIONS/END SECTIONS** directives and **SECTIONS** construct
  - Conditional compilation
  - Full OpenMP support for the **REDUCTION** clause in directives
  - Common blocks in data attribute scope clauses
  - **-qswapomp** compiler option, to indicate that the compiler should recognize and substitute OpenMP routines that exist in XL Fortran programs

Highlighting Conventions

The highlighting conventions have changed for the XLF Version 7.1 version of this document. Fortran 90 material is no longer differentiated from FORTRAN 77 material, and the highlighting colors for Fortran 95 and XL Fortran extensions have changed. For more information, see “Highlighting Conventions” on page xiv.

How to Use This Book

This book is intended primarily as a reference tool, not as a Fortran or programming tutorial. Programmers using this book should have some knowledge of Fortran concepts and have previous experience in writing Fortran application programs.

The book is divided into three main parts:

- *Concepts and Elements* provides some general background on the items and constructs you can use to write XL Fortran programs. See "Contents" on page iii for details on how this information is arranged.
- *Statements and Directives* provides an alphabetical reference of all XL Fortran statements. This section also provides reference information on both comment form and noncomment form directives, including the symmetric multiprocessing (SMP) directives.
  - Each statement description includes a brief explanation of purpose, a syntax diagram, a detailed discussion of rules concerning the statement, examples, and references to any related information. This section is particularly useful to programmers who know which statement they need, but want details on syntax or rules.
  - Directive descriptions are organized in a manner similar to the *Statements* section described above. It is important to gain an understanding of the directives contained in this section before programming using SMP.
- *Intrinsic and Library Procedures* provides an alphabetical reference to all XL Fortran:
  - Intrinsic procedures
  - Service and utility procedures
  - Pthreads library module

"Appendix A. Compatibility Across Standards" on page 731 discusses upwards and backwards compatibility across the FORTRAN 77, Fortran 90 and Fortran 95 standards. "Appendix B. ASCII and EBCDIC Character Sets" on page 737 displays the ASCII and EBCDIC character sets. The "Glossary" on page 751 provides an alphabetical reference to terms commonly used in this document. See "Highlighting Conventions" on page xiv for details on the use of color in this document.

© Copyright IBM Corp. 1990, 2000

Highlighting Conventions

This book uses the following highlighting conventions:

- Fortran 95 material that has been added to Fortran 90 is highlighted in red.
- Extensions to the Fortran 90 or Fortran 95 standards are highlighted in blue. These include:
  1. Any XL Fortran implementation of what the standards documents describe as a processor-dependent value, or processor-dependent behavior
  2. Any information added to XL Fortran Version 6.1 for XL Fortran Version 7.1 that is not part of the Fortran 90 or Fortran 95 standards

Fortran keywords, commands, statements, directives, intrinsic procedures, compiler options, and filenames are shown in bold: for example, **OPEN**, **COMMON**, and **END**.

References to other sources of information appear in italics. Variable names and user-specified names appear in lowercase italics: for example, *array_element_name*.

Note: Not all material throughout this book that mentions a feature belonging to either the new standards category or the extension category is highlighted; only the primary description of a given feature is. For example, the description of the **ALLOCATABLE** statement (see "ALLOCATABLE" on page 245) is highlighted, but not all of the rest of the references to the **ALLOCATABLE** statement that appear throughout the book are.

How to Read Syntax Diagrams

Throughout this book, the syntax of Fortran statements and elements is illustrated by diagrams, using notation often called "railroad tracks".

If a variable or user-specified name ends in *_list*, it means that you can provide a list of these terms separated by commas.

Punctuation marks, parentheses, arithmetic operators, and other special characters must be entered as part of the syntax.

Syntax Diagrams

- Syntax diagrams are read from left to right and from top to bottom, following the path of the line:

  The ►►── symbol indicates the beginning of a statement.

  The ───► symbol indicates that the statement syntax is continued on the next line.

The ►—— symbol indicates that a statement is continued from the previous line.
The ——►◄ symbol indicates the end of a statement.
Program units, procedures, constructs, interface blocks and derived-type definitions consist of several individual statements. For such items, a box encloses the syntax representation, and individual syntax diagrams show the required order for the equivalent Fortran statements.

- Required items appear on the horizontal line (the main path):

```
►►—keyword—required_argument————————————————————————————————————►◄
```

- Optional items appear below the main path:

```
►►—keyword—┬———————————————————┬————————————————————————————————►◄
           └—optional_argument—┘
```

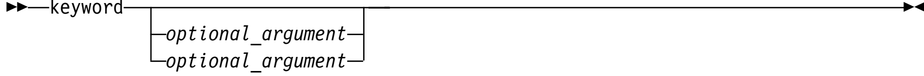
Note: Optional items (not in syntax diagrams) are enclosed by square brackets ([and]). For example, `[UNIT=]u`

- If you can choose from two or more items, they appear vertically, in a stack.

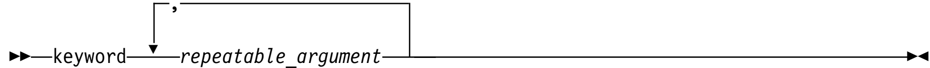
  If you *must* choose one of the items, one item of the stack appears on the main path:

```
►►—keyword——┬—required_argument—┬———————————————————————————————►◄
            └—required_argument—┘
```

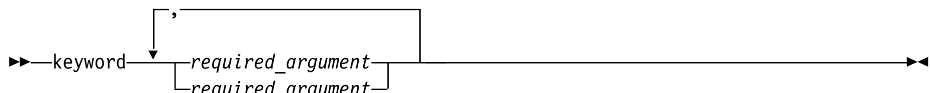
If choosing one of the items is optional, the entire stack appears below the main path:



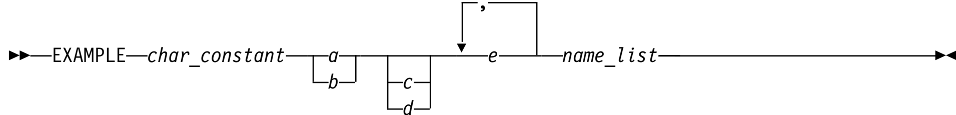
- An arrow returning to the left above the main line (a repeat arrow) indicates an item that can be repeated, and the separator character if it is other than a blank:



A repeat arrow above a stack indicates that you can make more than one choice from the stacked items.



Example of a Syntax Diagram



Interpret the diagram as follows:

- Enter the keyword **EXAMPLE**.
- Enter a value for *char_constant*.
- Enter a value for *a* or *b*, but not for both.
- Optionally, enter a value for *c* or *d*.
- Enter at least one value for *e*. If you enter more than one value, you must put a comma between each.
- Enter the value of at least one *name* for *name_list*. If you enter more than one value, you must put a comma between each. (The *_list* syntax is equivalent to the previous syntax for *e*.)

Notes on the Examples in This Book

- The examples in this book are coded in a simple style that does not try to conserve storage, check for errors, achieve fast performance, or demonstrate all possible ways to do something.
- The examples in this book are compiled using one of these invocation commands: **f77**, **xlf**, **xlf_r**, **xlf_r7**, **xlf90**, **xlf90_r**, **xlf90_r7**, **xlf95**, **xlf95_r**, **xlf95_r7**. See "Compiling an XL Fortran Program" in the *User's Guide* for details.
- You can paste the sample code from an HTML session into an edit session.
- Some sample programs from this book, and some other programs that illustrate ideas presented in this book, are in the directory **/usr/lpp/xlf/samples**.

Related Documentation

You might want to refer to the following publications for additional information:

- *XL Fortran for AIX User's Guide*, describes how to compile, link, and run your Fortran programs using XL Fortran Version 7.1.

- *Engineering and Scientific Subroutine Library Guide and Reference*, describes the Engineering and Scientific Subroutine Library (ESSL) routines.
- *AIX Technical Reference: Base Operating System and Extensions Volume 1* and *AIX Technical Reference: Base Operating System and Extensions Volume 2*, describe AIX subroutines, system calls, and Basic Linear Algebra Subroutines (BLAS).
- *Program Builder User's Guide* and *LPEX Editor User's Guide* explain some of the Common Desktop Environment tools that are integrated with XL Fortran.

Standards Documents

XL Fortran is designed according to the following standards documents. You may want to refer to these standards for precise definitions of some of the features referred to in this book.

1. *American National Standard Programming Language FORTRAN*, ANSI X3.9-1978.
2. *American National Standard Programming Language Fortran 90*, ANSI X3.198-1992. (This book uses its informal name, Fortran 90. This standard is equivalent to the ISO standard listed in point 5 below.)
3. *ANSI/IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985.
4. *High Performance Fortran Language Specification*, Version 1.1. CRPC=TR92225, Center for Research on Parallel Computation, Rice University, Houston, 1994.
5. *Information technology - Programming languages - Fortran*, ISO/IEC 1539-1:1991 (E).
6. *Information technology - Programming languages - Fortran - Part 1: Base language*, ISO/IEC 1539-1:1997. (This book uses its informal name, Fortran 95.)
7. *Military Standard Fortran DOD Supplement to ANSI X3.9-1978*, MIL-STD-1753 (United States of America, Department of Defence standard). Note that XL Fortran supports only those extensions documented in this standard that have also been subsequently incorporated into the Fortran 90 standard.
8. *OpenMP Fortran Language Application Program Interface*, Version 1.0 (Oct 1997) specification.

Part 1. Introduction

Part 1 gives a brief description of the Fortran language as it applies to the XL Fortran compiler implementation.

© Copyright IBM Corp. 1990, 2000

Chapter 1. Introduction to XL Fortran for AIX

This chapter describes:
- "XL Fortran (XLF)"
- "What is OpenMP?" on page 4
- "What is Fortran 95?" on page 5
- "What is Fortran 90?" on page 6
- "Valid and Invalid XL Fortran Programs" on page 9

This book describes Version 7.1 of the XL Fortran compiler. Fortran (FORmula TRANslation) is a high-level programming language primarily designed for applications involving numeric computations. It is suitable for most scientific, engineering, and mathematical applications.

XL Fortran is an implementation of the International Organization for Standardization (ISO) standard for Fortran 95, equivalent to the American National Standards Institute (ANSI) standard for Fortran 95. XL Fortran also implements other selected industry standards. See "Standards Documents" on page xvii.

The *User's Guide* discusses how to compile, link, and run programs written for XL Fortran. See "Related Documentation" on page xvi

Tool Tip

We would especially like to draw your attention to the commands **/usr/bin/xldb** and **/usr/bin/idebug** for debugging and **/usr/bin/xxlf** for selecting compiler options. The **/usr/bin/idebug** command launches the IBM Distributed Debugger.

XL Fortran (XLF)

The XLF language contains:
- Fortran 90

  The Fortran 90 language standard supports the FORTRAN 77 language standard. See "Appendix A. Compatibility Across Standards" on page 731 for more details.
- Fortran 95

  Fortran 95 has removed a few features that were part of both the Fortran 90 and FORTRAN 77 language standards. The features that have been removed from the Fortran 95 standard were deemed to conform least with

© Copyright IBM Corp. 1990, 2000

modern day programming practice by the Fortran standards committees. However, functionality has not been removed from Fortran 95 as efficient alternatives to the features deleted do exist. See "Deleted Features" on page 735 for more information. XL Fortran continues to support the features that have been removed from the Fortran 95 standard, since it is intended to be compatible with the FORTRAN 77, Fortran 90 and Fortran 95 standards.

Fortran 95 also provides an extensive set of additional language features. Some of these features are briefly discussed in the following section. The Fortran standard committees are responding to questions of interpretation about aspects of Fortran. Some of these questions may be related to language features that have been implemented in the XL Fortran compiler. Any answers given by these committees that are related to these language features may result in changes in future releases of the XL Fortran compiler, even if these changes result in incompatibilities with previous releases of the product.

- IBM and industry extensions to the Fortran language

  XL Fortran supports a number of extensions to the FORTRAN 77, Fortran 90, and Fortran 95 language standards. These extensions include, among other things, support for the OpenMP specification, support for a series of service and utility procedures and support for the Pthreads Library Module. Some of these functions are discussed briefly in the next section.

What is OpenMP?

OpenMP is a specification for a portable implementation of shared memory parallelism in Fortran and C/C++. The OpenMP specification provides a number of compiler directives, library routines and environment variables that you can use to control the parallel environment. This section gives a brief overview of the features of OpenMP.

Compiler Directives

The OpenMP Application Program Interface provides a number of directives that define parallel regions, work-sharing constructs and synchronization constructs, and provide support for the sharing and privatization of data.

See "Chapter 11. Directives" on page 423 for more information.

Library Routines

The OpenMP API supports a suite of library routines that allow you to control and query the run-time execution environment, and a set of general-purpose locking routines.

See "Chapter 14. OpenMP Execution Environment Routines and Lock Routines" on page 667 for more information.

Environment Variables

The OpenMP environment variables also provide you with the functionality to control the run-time execution environment.

See the *User's Guide* for more information.

What is Fortran 95?

The Fortran 95 language standard contains most features that were part of both the Fortran 90 and FORTRAN 77 language standards. However, Fortran 95 has removed a few features that were part of both the Fortran 90 and FORTRAN 77 language standards. See "Deleted Features" on page 735 for more information. The main improvements offered by the Fortran 95 standard are:

FORALL

Fortran 95 provides an efficient alternative to the element by element construction of an array value in Fortran 90. The **FORALL** statement allows you to explicitly specify array elements, array sections, character substrings, or pointer targets as a function of the element subscripts.

See "FORALL" on page 311 for more information.

PURE

Fortran 95 functions may have side effects. Side effects that may occur include a change in value of an argument or global variable. To avoid problems associated with side effects, Fortran 95 enables you to specify a function as side effect free. These functions are referred to as **PURE** functions. A restricted form of a **PURE** procedure is called **ELEMENTAL**.

See "Pure Procedures" on page 176 for more information.

ELEMENTAL

Fortran 95 provides an efficient alternative to the problem of providing a procedure that operates elementally. In Fortran 90, to provide a procedure that operated elementally you had to produce eight versions of the procedure: one to operate on scalars, and one for each rank of the array from one to seven. If you used two array arguments as many as sixteen versions could be required. Using **ELEMENTAL**, you can write a single procedure with the same functionality.

See "Elemental Procedures" on page 178 for more information.

Initialization

Fortran 95 provides an improvement over Fortran 90 in that it provides a means of defining the initial pointer association status of a pointer. Fortran 95 provides an initial pointer association status other than undefined through the use of the **NULL** intrinsic function.

See "NULL(MOLD)" on page 599 for more information.

In addition, Fortran 95 provides a means of specifying default initial values for derived-type components which enhances Fortran's data abstraction capabilities.

See "Derived Types" on page 39 for more information.

Specification Functions

Fortran 95 provides an improvement over Fortran 90 in that it allows user defined functions in specification expressions. The specification functions have a requirement that they be pure. This prevents them from having side affects that could affect other objects being declared in the same *specification_part*.

See "Specification Expressions" on page 88 for more information.

Deleted Features

Fortran 95 has removed a few features that were part of both the Fortran 90 and FORTRAN 77 language standards. The features that have been removed from the Fortran 95 standard were deemed to conform least with modern day programming practice by the Fortran standards committees. However, functionality has not been removed from Fortran 95 as efficient alternatives to the features deleted do exist.

XL Fortran continues to support the features that have been removed from the Fortran 95 standard.

See "Deleted Features" on page 735 for more information.

What is Fortran 90?

Fortran 90 adds a wealth of new features to FORTRAN 77. Fortran 90 may offer an improved method or feature above FORTRAN 77 for performing a given task. The following topics outline some of the key features that Fortran 90 brings to the FORTRAN 77 language.

Fortran 90 Free Source Form

In addition to the fixed source form format (defined in FORTRAN 77), Fortran 90 defines a free source form format. A statement can begin in any column, and blanks are significant.

Note: Because XL Fortran also defines an IBM free source form, the free source form defined by Fortran 90 is called Fortran 90 free source form in this book.

See “Free Source Form” on page 19 for more information.

Parameterized Data Types

Although the length specification for a data type (for example, INTEGER*4) is a common industry extension, Fortran 90 provides facilities for specifying the precision and range of noncharacter intrinsic data types and the character sets available for the character data type. When used with the **SELECTED_INT_KIND** and **SELECTED_REAL_KIND** intrinsic functions, data types that use parameters become portable across platforms.

See “Type Parameters and Specifiers” on page 27 for more information.

Derived Types

A derived type is a user-defined type with components of intrinsic type and/or other derived types. Objects of a derived type can be used in intrinsic assignment, in input/output, and as procedure arguments. When used with defined or extended intrinsic operations, derived types can be used to provide powerful data abstractions (for example, linked lists).

See “Derived Types” on page 39 for more information.

Array Enhancements

With Fortran 90, you can specify array expressions and assignments. An array section, a portion of a whole array, can be used as an array. Array constructors offer a concise syntax for specifying the values of an array. Assumed-shape arrays, deferred-shape arrays, and automatic arrays give you more flexibility when you use arrays. Use the **WHERE** construct to mask array expression evaluation and array assignment.

See “Chapter 4. Array Concepts” on page 63 for more information.

Pointers

Pointers refer to memory addresses instead of values. Pointers provide the means for creating linked lists and dynamic arrays. Objects of an intrinsic or a derived-type can be declared to be pointers.

See “Pointer Association” on page 139 and “POINTER (Fortran 90)” on page 364 for more information.

Dynamic Behavior

Storage is not set aside for pointer targets and allocatable arrays at compile time. The **ALLOCATE** and **DEALLOCATE** statements let you control the

storage usage at run time. You can also use pointer assignment to alter the storage space associated with the pointer.

See “ALLOCATE” on page 246, “DEALLOCATE” on page 280, and “Pointer Assignment” on page 117 for more information.

Control Construct Enhancements

The **CASE** construct provides a concise syntax for selecting, at most, one of a number of statement blocks for execution. The case expression of each **CASE** block is evaluated against that of the construct.

The **DO** statement with no control clause and the **DO WHILE** construct offer increased versatility. In addition, the **CYCLE** and **EXIT** statements provide control over the execution of the construct from within the construct.

Control constructs can be given names, which aids readability and syntax-checking for nested constructs.

See “Chapter 6. Control” on page 121 for more information.

Procedure Enhancements

Fortran 90 introduces many new features that make the use of procedures easier. Functions can extend intrinsic operators and define new operators. With subroutines, you can extend intrinsic assignments. The actual arguments of a procedure can be specified with keywords, and you can explicitly indicate the intended use of dummy arguments and whether or not they are optional.

The interface or characteristics of a dummy or external procedure can be explicitly specified in an interface block. A generic interface block can specify a name that can be referenced to access any specific procedures defined within the block, depending on the nature of the actual arguments.

Internal procedures, contained within a main program or another subprogram, let you partition programs while allowing you to access entities defined in the host procedure.

Recursive procedures are allowed in Fortran 90; functions and subroutines can call themselves directly or indirectly.

See “Chapter 7. Program Units and Procedures” on page 133 for more information.

Modules

Modules provide the means for data encapsulation and the operations that apply to the data. A module is a nonexecutable program unit that can contain data object declarations, derived-type definitions, procedures, and procedure

interfaces. With modules, you can specify that some entities can be used only within the module, and other entities can be accessed from any program unit.

See "Modules" on page 153 for more information.

New Intrinsic Procedures

Fortran 90 brings dozens of new intrinsic procedures to Fortran. For example, a set of transformational intrinsic procedures provides powerful array manipulation capabilities. Many new inquiry functions enable you to examine the properties of entities.

See "Chapter 12. Intrinsic Procedures" on page 519 for details.

Valid and Invalid XL Fortran Programs

This book defines the syntax, semantics, and restrictions you must follow to write valid XL Fortran programs. The compiler discovers most nonconformances of the XL Fortran language rules, but it may not find some syntactic and semantic combinations. This occurs because of either performance reasons, or because the nonconformance is only detectable at run time. XL Fortran programs that contain these undiagnosed combinations are not valid, whether or not they run as expected.

Part 2. Concepts and Elements

Part 2 provides information on:

- "Chapter 2. The Language Elements" on page 13
- "Chapter 3. Data Types and Data Objects" on page 27
- "Chapter 4. Array Concepts" on page 63
- "Chapter 5. Expressions and Assignment" on page 85
- "Chapter 6. Control" on page 121
- "Chapter 7. Program Units and Procedures" on page 133
- "Chapter 8. Input/Output Concepts" on page 183
- "Chapter 9. Input/Output Formatting" on page 201

© Copyright IBM Corp. 1990, 2000

Chapter 2. The Language Elements

This chapter describes the elements of an XL Fortran program:
- "Characters"
- "Names" on page 14
- "Statements" on page 15
- "Lines and Source Formats" on page 16
- "Order of Statements and Execution Sequence" on page 25

Characters

The XL Fortran character set consists of letters, digits, and special characters:

Letters	Digits	Special Characters
A N a n	0	Blank
B O b o	1	= Equal sign
C P c p	2	+ Plus sign
D Q d q	3	- Minus sign
E R e r	4	* Asterisk
F S f s	5	/ Slash
G T g t	6	(Left parenthesis
H U h u	7	) Right parenthesis
I V i v	8	, Comma
J W j w	9	. Decimal point / period
K X k x		$ Currency symbol
L Y l y		' Apostrophe
M Z m z		: Colon
		! Exclamation point
		" Double quotation mark
		% Percent sign
		& Ampersand
		; Semicolon
		? Question mark
		< Less than
		> Greater than
		_ Underscore

The characters have an order known as a *collating sequence*, which is the arrangement of characters that determines their sequence order for such processes as sorting, merging, comparing. XL Fortran uses American National Standard Code for Information Interchange (ASCII) to determine the ordinal sequence of characters. (See "Appendix B. ASCII and EBCDIC Character Sets" on page 737 for a complete listing of the ASCII character set.)

© Copyright IBM Corp. 1990, 2000

White space refers to blanks and tabs. The significance of white space depends on the source format used. See "Lines and Source Formats" on page 16 for details.

A *lexical token* is a sequence of characters with an indivisible interpretation that forms a building block of a program. It can be a keyword, name, literal constant (not of type complex), operator, label, delimiter, comma, equal sign, colon, semicolon, percent sign, `::`, or =>.

Names

A *name* is a sequence of any or all of the following elements:
- Letters (A-Z, a-z)
- Digits (0-9)
- Underscores (_)
- Dollar signs ($)

The first character of a name must not be a digit.

In Fortran 90 and Fortran 95, the maximum length of a name is 31 characters.

In XL Fortran, the maximum length of a name is 250 characters. Although XL Fortran allows a name to start with an underscore, you may want to avoid using one in that position because the AIX operating system, and the XL Fortran compiler and libraries have reserved names that begin with underscores.

All letters in a source program are translated into lowercase unless they are in a character context. The character contexts are characters within character literal constants, character-string edit descriptors, and Hollerith constants.

Note: If you specify the **-qmixed** compiler option, names are not translated to lowercase. For example, XL Fortran treats

```
ia Ia iA IA
```

the same by default, but treats them distinctly if you specify the **-qmixed** compiler option.

A name can identify entities such as:
- A variable
- A constant
- A procedure
- A derived type
- A construct
- A **CRITICAL** construct
- A program unit
- A common block

- A namelist group

A subobject designator is a name followed by one or more selectors (array element selectors, array section selectors, component selectors, and substring selectors). It identifies the following items in a program unit:
- An array element (see “Array Elements” on page 73)
- An array section (see “Array Sections” on page 74)
- A structure component (see “Structure Components” on page 43)
- A character substring (see “Character Substrings” on page 37)

Statements

A Fortran statement is a sequence of lexical tokens. Statements are used to form program units.

The maximum length of a statement in XL Fortran is 6700 characters.

See “Part 3. Statements and Directives” on page 239 of this book for details on all statements supported by XL Fortran.

Statement Keywords

A statement keyword is part of the syntax of a statement, and appears in uppercase bold everywhere but in syntax diagrams and tables. For example, the term **DATA** in the **DATA** statement is a statement keyword.

No sequence of characters is reserved in all contexts. A statement keyword is interpreted as an entity name if the keyword is used in such a context.

Statement Labels

A statement label is a sequence of one to five digits, one of which must be nonzero, that you can use to identify statements in a Fortran scoping unit. In fixed source form, a statement label can appear anywhere in columns 1 through 5 of the initial line of the statement. In free source form, such column restrictions do not apply.

XL Fortran ignores all characters that appear in columns 1 through 5 on fixed source form continuation lines.

Giving the same label to more than one statement in a scoping unit will cause ambiguity, and the compiler will generate an error. White space and leading zeros are not significant in distinguishing between statement labels. You can label any statement, but statement labels can only refer to executable statements and **FORMAT** statements. The statement making the reference and the statement it references (identified by the statement label) must be in the same scoping unit in order for the reference to resolve. (See “Scope” on page 133 for details).

Lines and Source Formats

A line is a horizontal arrangement of characters. By contrast, a column is a vertical arrangement of characters, where each character, or each byte of a multibyte character, in a given column shares the same line position.

Because XL Fortran measures lines in bytes, these definitions apply only to lines containing single-byte characters. Each byte of a multibyte character occupies one column.

The kinds of lines are:

Initial Line	Is the first line of a statement.
Continuation Line	Continues a statement beyond its initial line.
Comment Line	Does not affect the executable program and can be used for documentation. The comment text continues to the end of a line. Although comment lines can follow one another, a comment line cannot be continued. A line of all white space or a zero-length line is a comment line without any text. Comment text can contain any characters allowed in a character context. If an initial line or continuation line is not continued, or if it is continued but not in a character context, an inline comment can be placed on the same line, to the right of any statement label, statement text, and continuation character that may be present. An exclamation mark (!) begins an inline comment.
Conditional Compilation Line	Indicates that the line should only be compiled if recognition of conditional compilation lines is enabled. A conditional compilation sentinel should appear on a conditional compilation line. (See "Conditional Compilation" on page 22)
Debug Line	Indicates that the line is for debugging code (for fixed source form only). The letter `D` must be specified in column 1. (See "Debug Lines" on page 18)
Directive Line	Provides instructions or information to the compiler (see "Chapter 11. Directives" on page 423).

Source lines can be in fixed source form or free source form format. Use the **SOURCEFORM** directive to mix source formats within the same program unit. Fixed source form is the default when using the **f77**, **xlf**, **xlf_r** or **xlf_r7** invocation commands. Fortran 90 free source form is the default when using the **xlf90**, **xlf90_r**, **xlf90_r7**, **xlf95**, **xlf95_r** or **xlf95_r7** invocation commands.

See "Compiling an XL Fortran Program" in the *User's Guide* for details on invocation commands.

Fixed Source Form

A fixed source form line is a sequence of 1 to 132 characters. The default line size (as stipulated in Fortran 95) is 72 characters, but can be changed using the **-qfixed**=*right_margin* compiler option (see the *User's Guide*).

Columns beyond the right margin are not part of the line and can be used for identification, sequencing, or any other purpose.

Except within a character context, white space is insignificant; that is, you can imbed white space between and within lexical tokens, without affecting the way the compiler will treat them.

Tab formatting means there is a tab character in columns 1 through 6 of an initial line, which directs the compiler to interpret the next character as being in column 7.

Requirements for lines and for items on those lines are:

- A comment line begins with a `C`, `c`, or an asterisk (`*`) in column 1, or is all white space. Comments can also follow an exclamation mark (!), except when the exclamation mark is in column 6 or in a character context.
- For an initial line without tab formatting:
  - Columns 1 through 5 contain either blanks, a statement label, or a D in column 1 optionally followed by a statement label.
  - Column 6 contains a blank or zero.
  - Columns 7 through to the right margin contain statement text, possibly followed by other statements or by an inline comment.
- For an initial line with tab formatting:
  - Columns 1 through 6 begin with either blanks, a statement label, or a D in column 1 optionally followed by a statement label. This must be followed by a tab character.
  - If the **-qxflag=oldtab** compiler option is specified, all columns from the column immediately following the tab character through to the right margin contain statement text, possibly followed by other statements and by an inline comment.
  - If the **-qxflag=oldtab** compiler option is not specified, all columns from column 7 (which corresponds to the character after the tab) to the right margin contain statement text, possibly followed by other statements and by an inline comment.
- For a continuation line:
  - Column 1 must not contain `C`, `c`, or an asterisk. Columns 1 through 5 must not contain an exclamation mark as the leftmost nonblank character.

Column 1 can contain a D (signifying a debug line). Otherwise, these columns can contain any characters allowed in a character context; these characters are ignored.
  - Column 6 must have either a nonzero character or white space. The character in column 6 is referred to as the continuation character. Exclamation marks and semicolons are valid continuation characters.
  - Columns 7 through to the right margin contain continued statement text, possibly followed by other statements and an inline comment.
  - Neither the **END** statement nor a statement whose initial line appears to be a program unit **END** statement can be continued.
  - There is no limit to the number of continuation lines for a statement, but a statement cannot be longer than 6700 characters.

    The Fortran standards limit the number of continuation lines to 19.

A semicolon (;) separates statements on a single source line, except when it appears in a character context, in a comment, or in columns 1 through 6. Two or more semicolon separators that are on the same line and are themselves separated by only white space or other semicolons are considered to be a single separator. A separator that is the last character on a line or before an inline comment is ignored. Statements following a semicolon on the same line cannot be labeled. Additional statements cannot follow a program unit **END** statement on the same line.

Debug Lines

A debug line, allowed only for fixed source form, contains source code used for debugging and is specified by the letter D in column 1. The handling of debug lines depends on the **-qdlines** compiler option:

- If you specify the **-qdlines** option, the compiler interprets the D in column 1 as a blank, and handles such lines as lines of source code.
- If you do not specify **-qdlines**, the compiler handles such lines as comment lines. This is the default setting.

If you continue a debugging statement on more than one line, every continuation line must have a D in column 1 and a continuation character. If the initial line is not a debugging line, you can designate any continuation lines as debug lines provided that the statement is syntactically correct, whether or not you specify the **-qdlines** option.

Example of Fixed Source Form:

```
C Column Numbers:
C        1         2         3         4         5         6         7
C23456789012345678901234567890123456789012345678901234567890123456789012

!IBM* SOURCEFORM (FIXED)
      CHARACTER ABC ; LOGICAL X          ! 2 statements on 1 line
      DO 10 I=1,10
```

```
        PRINT *,'this is the index',I  ! with an inline comment
10    CONTINUE
C
       CHARSTR="THIS IS A CONTINUED
     X CHARACTER STRING"
       ! There will be 38 blanks in the string between "CONTINUED"
       ! and "CHARACTER". You cannot have an inline comment on
       ! the initial line because it would be interpreted as part
       ! of CHARSTR (character context).
  100 PRINT *, IERROR
! The following debug lines are compiled as source lines if
! you use -qdlines
D     IF (I.EQ.IDEBUG.AND.
D    +    J.EQ.IDEBUG)     WRITE(6,*) IERROR
D     IF (I.EQ.
D    +  IDEBUG )
D    +  WRITE(6,*) INFO
       END
```

Free Source Form

XL Fortran allows any line length and number of continuation lines, so long as the number of characters does not exceed 6700.

A free source form line can specify up to 132 characters on each line, with a maximum of 39 continuation lines for a statement.

Items can begin in any column of a line, subject to the following requirements for lines and items on those lines:

- A comment line is a line of white space or begins with an exclamation mark (!) that is not in a character context.
- An initial line can contain any of the following items, in the following sequence:
  - A statement label
  - Statement text. Note that statement text is required in an initial line.
  - Additional statements
  - The ampersand continuation character (&).
  - An inline comment
- If you want to continue an initial line or continuation line in a non-character context, the continuation line must start on the first noncomment line that follows the intial line or continuation line. To define a line as a continuation line, you must place an ampersand after the statements on the previous non-comment line.
- White space before and after the ampersand is optional, with the following restrictions:
  - If you also place an ampersand in the first nonblank character position of the continuation line, the statement continues at the next character position following the ampersand.

– If a lexical token is continued, the ampersand must immediately follow the initial part of the token, and the remainder of the token must immediately start after the ampersand on the continuation line.

- A character context can be continued if the following conditions are true:
  – The last character of the continued line is an ampersand and is not followed by an inline comment. If the rightmost character of the statement text to be continued is an ampersand, a second ampersand must be entered as a continuation character.
  – The first nonblank character of the next noncomment line is an ampersand.

A semicolon separates statements on a single source line, except when it appears in a character context or in a comment. Two or more separators that are on the same line and are themselves separated by only white space or other semicolons are considered to be a single separator. A separator that is the last character on a line or before an inline comment is ignored. Additional statements cannot follow a program unit **END** statement on the same line.

White Space

White space must not appear within lexical tokens, except in a character context or in a format specification. White space can be inserted freely between tokens to improve readability, although it must separate names, constants, and labels from adjacent keywords, names, constants, and labels.

Certain adjacent keywords may require white space. Table 1 on page 21 lists keywords that require white space, and keywords for which white space is optional.

Table 1. Keywords Where White Space is Optional

White Space Optional
BLOCK DATA
DOUBLE COMPLEX
DOUBLE PRECISION
ELSE IF
END BLOCK DATA
END DO
END FILE
END FORALL
END FUNCTION
END IF
END INTERFACE
END MODULE
END PROGRAM
END SELECT
END SUBROUTINE
END TYPE
END WHERE
GO TO
IN OUT
SELECT CASE

See "Type Declaration" on page 402 for details about *type_spec*.

Example of Free Source Form:

```
!IBM* SOURCEFORM (FREE(F90))
!
! Column Numbers:
!        1         2         3         4         5         6         7
!23456789012345678901234567890123456789012345678901234567890123456789012
 DO I=1,20
   PRINT *,'this statement&
   & is continued' ; IF (I.LT.5) PRINT *, I

 ENDDO
 EN&
         &D                ! A lexical token can be continued
```

IBM Free Source Form

An IBM free source form line or statement is a sequence of up to 6700 characters. Items can begin in any column of a line, subject to the following requirements for lines and items on those lines:

- A comment line begins with a double quotation mark (") in column 1, is a line of all white space, or is a zero-length line. A comment line must not follow a continued line. Comments can also follow an exclamation mark (!), except in a character context.

- An initial line can contain any of the following items, in the following sequence:
  - A statement label
  - Statement text
  - The minus sign continuation character (-)
  - An inline comment
- A continuation line immediately follows a continued line and can contain any of the following items, in the following sequence:
  - Statement text
  - A continuation character (-)
  - An inline comment

If statement text on an initial line or continuation line is to be continued, a minus sign indicates continuation of the statement text on the next line. In a character context, if the rightmost character of the statement text to be continued is a minus sign, a second minus sign must be entered as a continuation character.

Except within a character context, white space is insignificant; that is, you can imbed white space between and within lexical tokens, without affecting the way the compiler will treat them.

Example of IBM Free Source Form

```
!IBM* SOURCEFORM (FREE(IBM))
"
" Column Numbers:
"        1         2         3         4         5         6         7
"23456789012345678901234567890123456789012345678901234567890123456789012
 DO I=1,10
  PRINT *,'this is -
             the index',I   ! There will be 14 blanks in the string
                            ! between "is" and "the"
 END DO
 END
```

Conditional Compilation

You can use sentinels to mark specific lines of an XL Fortran program for conditional compilation. This support allows you to port code that contains statements that are only valid or needed in an SMP environment to a non-SMP environment. You can do this by using conditional compilation lines or by using the **_OPENMP** C preprocessor macro.

The syntax for conditional compilation lines is as follows:

```
►►—cond_comp_sentinel—fortran_source_line——————————————————————————►◄
```

cond_comp_sentinel
: is a conditional compilation sentinel that is defined by the current source form and is either:
  - **!$**, **C$**, **c$**, or ***$**, for fixed source form; or
  - **!$**, for free source form

fortran_source_line
: is an XL Fortran source line

The syntax rules for conditional compilation lines are very similar to the syntax rules for fixed source form and free source form lines. The rules are as follows:

- General Rules:

  A valid XL Fortran source line must follow the conditional compilation sentinel.

  A conditional compilation line may contain the **INCLUDE** or **EJECT** noncomment directives.

  A conditional compilation sentinel must not contain embedded white space.

  A conditional compilation sentinel must not follow a source statement or directive on the same line.

  If you are continuing a conditional compilation line, the conditional compilation sentinel must appear on at least one of the continuation lines or on the initial line.

  You must specify the **-qcclines** compiler option for conditional compilation lines to be recognized. To disable recognition of conditional compilation lines, specify the **-qnocclines** compiler option. Specifying the **-qsmp=omp** compiler option enables the **-qcclines** option.

  Trigger directives take precedence over conditional compilation sentinels. For example, if you specify the **-qdirective='$'** option, then lines that start with the trigger, such as **!$**, will be treated as comment directives, rather than conditional compilation lines.

- Fixed Source Form Rules:

  Conditional compilation sentinels must start in column 1.

  All of the rules for fixed source form line length, case sensitivity, white space, continuation, tab formatting, and columns apply. See "Fixed Source Form" on page 17 for information. Note that when recognition of conditional compilation lines is enabled, the conditional compilation sentinel is replaced by two white spaces.

- Free Source Form Rules:

  Conditional compilation sentinels may start in any column.

  All of the rules for free source form line length, case sensitivity, white space, and continuation apply. See "Free Source Form" on page 19 for

information. Note that when recognition of conditional compilation lines is enabled, the conditional compilation sentinel is replaced by two white spaces.

Another way to conditionally include code, other than using conditional compilation lines, is to use the C preprocessor macro **_OPENMP**. This macro is defined when the C preprocessor is invoked and you specify the **-qsmp=omp** compiler option. See the section on passing Fortran files through the C preprocessor in the "Editing, Compiling, Linking, and Running XL Fortran Programs" chapter of the *User's Guide* for an example of using this macro.

Valid Example of Conditional Compilation Lines

In the following example, conditional compilation lines are used to hide OpenMP run-time routines. Code that calls OpenMP run-time routines cannot easily be compiled in a non-OpenMP environment without using conditional compilation. Since calls to the run-time routines are not directives, they cannot be hidden by the !$OMP trigger. If the code below is not compiled with the **-qsmp=omp** compiler option, the variable used to store the number of threads will be assigned the value of 8.

```
      PROGRAM PAR_MAT_MUL
      IMPLICIT NONE
      INTEGER(KIND=8)                  ::I,J,NTHREADS
      INTEGER(KIND=8),PARAMETER        ::N=60
      INTEGER(KIND=8),DIMENSION(N,N)   ::AI,BI,CI
      INTEGER(KIND=8)                  ::SUMI
!$    INTEGER OMP_GET_NUM_THREADS

      COMMON/DATA/ AI,BI,CI
!$OMP THREADPRIVATE (/DATA/)

!$OMP PARALLEL
      FORALL(I=1:N,J=1:N) AI(I,J) = (I-N/2)**2+(J+N/2)
      FORALL(I=1:N,J=1:N) BI(I,J) = 3-((I/2)+(J-N/2)**2)
!$OMP MASTER
      NTHREADS=8
!$    NTHREADS=OMP_GET_NUM_THREADS()
!$OMP END MASTER
!$OMP END PARALLEL

!$OMP PARALLEL DEFAULT(PRIVATE),COPYIN(AI,BI),SHARED(NTHREADS)
!$OMP DO
      DO I=1,NTHREADS
      CALL IMAT_MUL(SUMI)
      ENDDO
!$OMP END DO
!$OMP END PARALLEL

      END
```

Order of Statements and Execution Sequence

Table 2. Statement Order

<table>
<tr><td colspan="2">1 PROGRAM, FUNCTION, SUBROUTINE, MODULE, or BLOCK DATA Statement</td></tr>
<tr><td colspan="2">2 USE Statements</td></tr>
<tr><td rowspan="2">3 DATA, FORMAT, and ENTRY Statements</td><td>4 Derived-Type Definitions, Interface Blocks, Type Declaration Statements, Specification Statements, IMPLICIT Statements, and PARAMETER Statements</td></tr>
<tr><td>5 Executable constructs</td></tr>
<tr><td colspan="2">6 CONTAINS Statement</td></tr>
<tr><td colspan="2">7 Internal Subprograms or Module Subprograms</td></tr>
<tr><td colspan="2">8 END Statement</td></tr>
<tr><td colspan="2">Statement Order

Vertical lines delineate varieties of statements that can be interspersed, while horizontal lines delineate varieties of statements that cannot be interspersed. The numbers in the diagram reappear later in the book to identify groups of statements that are allowed in particular contexts. A reference back to this section is included in the places where these numbers are used in the rest of this book.</td></tr>
</table>

Refer to "Chapter 7. Program Units and Procedures" on page 133 or "Chapter 10. Statements" on page 241 for more details on rules and restrictions concerning statement order.

Normal execution sequence is the processing of references to specification functions in any order, followed by the processing of executable statements in the order they appear in a scoping unit.

A transfer of control is an alteration of the normal execution sequence. Some statements that you can use to control the execution sequence are:

- Control statements
- Input/output statements that contain an **END=**, **ERR=**, or **EOR=** specifier

When you reference a procedure that is defined by a subprogram, the execution of the program continues with any specification functions referenced in the scoping unit of the subprogram that defines the procedure. The program resumes with the first executable statement following the **FUNCTION**, **SUBROUTINE** or **ENTRY** statement that defines the procedure. When you return from the subprogram, execution of the program continues from the point at which the procedure was referenced or to a statement referenced by an alternate return specifier.

In this book, any description of the sequence of events in a specific transfer of control assumes that no event, such as the occurrence of an error or the execution of a **STOP** statement, changes that normal sequence.

Chapter 3. Data Types and Data Objects

This chapter describes:

- "Data Types"
- "Data Objects" on page 28
- "Intrinsic Types" on page 29
- "Derived Types" on page 39
- "Typeless Literal Constants" on page 46
- "How Type Is Determined" on page 52
- "Definition Status of Variables" on page 53
- "Allocation Status" on page 58
- "Storage Classes for Variables" on page 59

Data Types

A data type has a name, a set of valid values, a means to denote such values (constants), and a set of operations to manipulate the values. There are two categories of data types: *intrinsic types* and *derived types*.

The intrinsic types, including their operations, are predefined and are always accessible. There are two classes of intrinsic data types:

- **Numeric (also known as Arithmetic):** integer, real, complex, and byte
- **Nonnumeric:** character, logical, and byte

Derived types are user-defined data types whose components are intrinsic and/or derived data types.

Type Parameters and Specifiers

XL Fortran provides one or more representation methods for each of the intrinsic data types. Each method can be specified by a value called a *kind type parameter*, which indicates the decimal exponent range for the integer type, the decimal precision and exponent range for the real and complex types, and the representation methods for the character and logical types. Each intrinsic type supports a specific set of kind type parameters. *kind_param* is either a *digit_string* or *scalar_int_constant_name*.

The *length type parameter* specifies the number of characters for entities of type character.

A *type specifier* specifies the type of all entities declared in a type declaration statement. Some type specifiers (**INTEGER**, **REAL**, **COMPLEX**, **LOGICAL**, and **CHARACTER**) can include a *kind_selector*, which specifies the *kind type parameter*.

© Copyright IBM Corp. 1990, 2000

For example, a 4-byte integer can be declared as **INTEGER(4)**, **INTEGER(KIND=4)**, **INTEGER*4**, or, if the default integer size is set to 4 bytes, simply **INTEGER**. In this book, references to 4-byte integers take the **INTEGER(4)** form. See *type_spec* on page 402 for details on using type specifiers.

The **KIND** intrinsic function returns the kind type parameter of its argument. See "KIND(X)" on page 574 for details.

Data Objects

A *data object* is a variable, constant, or subobject of a constant.

A *variable* can have a value and can be defined or redefined during execution of an executable program. A variable can be:
- A scalar variable name
- An array variable name
- A subobject

A *subobject* (of a variable) is a portion of a named object that can be referenced and defined. It can be:
- An array element
- An array section
- A character substring
- A structure component

A subobject of a constant is a portion of a constant. The referenced portion may depend on a variable value.

Constants

A *constant* has a value and cannot be defined or redefined during execution of an executable program. A constant with a name is a *named constant* (see "PARAMETER" on page 362). A constant without a name is a *literal constant*. A literal constant can be of intrinsic type or it can be typeless (hexadecimal, octal, binary, or Hollerith). The optional kind type parameter of a literal constant can only be a digit string or a scalar integer named constant.

A signed literal constant can have a leading plus or minus sign. All other literal constants must be unsigned; they must have no leading sign. The value zero is considered neither positive nor negative. You can specify zero as signed or unsigned.

Automatic Objects

An *automatic object* is a data object that is dynamically allocated within a procedure. It is a local entity of a subprogram and has a nonconstant character length and/or a nonconstant array bound. It is not a dummy argument.

An automatic object always has the controlled automatic storage class.

An automatic object cannot be specified in a **DATA**, **EQUIVALENCE**, **NAMELIST**, or **COMMON** statement, nor can the **AUTOMATIC**, **STATIC**, **PARAMETER**, or **SAVE** attributes be specified for it. An automatic object cannot be initialized or defined with an initialization expression in a type declaration statement, but it can have a default initialization. An automatic object cannot appear in the specification part of a main program or module.

Intrinsic Types

Integer

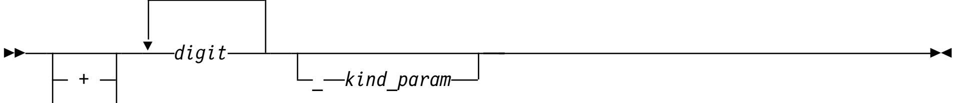
The following table shows the range of values that XL Fortran can represent using the integer data type:

Kind parameter	Range of values
1	-128 through 127
2	-32 768 through 32 767
4	-2 147 483 648 through 2 147 483 647
8	-9 223 372 036 854 775 808 through 9 223 372 036 854 775 807

XL Fortran sets the default kind type parameter to 4. The kind type parameter is equivalent to the byte size for integer values. Use the **-qintsize** compiler option to change the default integer size to 2, 4, or 8 bytes. Note that the **-qintsize** option similarly affects the default logical size.

The integer type specifier must include the **INTEGER** keyword. See “INTEGER” on page 338 for details on declaring entities of type integer.

The form of a signed integer literal constant is:



kind_param is either a *digit-string* or a *scalar-int-constant-name*

A signed integer literal constant has an optional sign, followed by a string of decimal digits containing no decimal point and expressing a whole number, optionally followed by a kind type parameter. A signed, integer literal constant can be positive, zero, or negative. If unsigned and nonzero, the constant is assumed to be positive.

If *kind_param* is specified, the magnitude of the literal constant must be representable within the value range permitted by that *kind_param*.

If no *kind_param* is specified and the magnitude of the constant cannot be represented as a default integer, the constant is promoted to a representable kind.

XL Fortran represents integers internally in two's-complement notation, where the leftmost bit is the sign of the number.

Examples of Integer Constants

```
0                        ! has default integer size
-173_2                   ! 2-byte constant
9223372036854775807      ! Kind type parameter is promoted to 8
```

Real

The following table shows the range of values that XL Fortran can represent with the real data type:

Kind Parameter	Approximate Absolute Nonzero Minimum	Approximate Absolute Maximum	Approximate Precision (decimal digits)
4	1.175494E-38	3.402823E+38	7
8	2.225074D-308	1.797693D+308	15
16	2.225074Q-308	1.797693Q+308	31

XL Fortran sets the default kind type parameter to 4. The kind type parameter is equivalent to the byte size for real values. Use the **-qrealsize** compiler option to change the default real size to 4 or 8 bytes. Note that the **-qrealsize** option affects the default complex size.

XL Fortran represents **REAL(4)** and **REAL(8)** numbers internally in the ANSI/IEEE binary floating-point format, which consists of a sign bit (s), a biased exponent (e), and a fraction (f). The **REAL(16)** representation is based on the **REAL(8)** format.

```
REAL(4)
Bit no. 0....|....1....|....2....|....3.
        seeeeeeeefffffffffffffffffffffff
REAL(8)
Bit no. 0....|....1....|....2....|....3....|....4....|....5....|....6...
        seeeeeeeeeeeffffffffffffffffffffffffffffffffffffffffffffffffffff
REAL(16)
Bit no. 0....|....1....|....2....|....3....|....4....|....5....|....6...
        seeeeeeeeeeeffffffffffffffffffffffffffffffffffffffffffffffffffff
Bit no. |....7....|....8....|....9....|....0....|....1....|....2....|..
        seeeeeeeeeeefffffffffffffffffffffffffffffffffffffffffffffffffffff
```

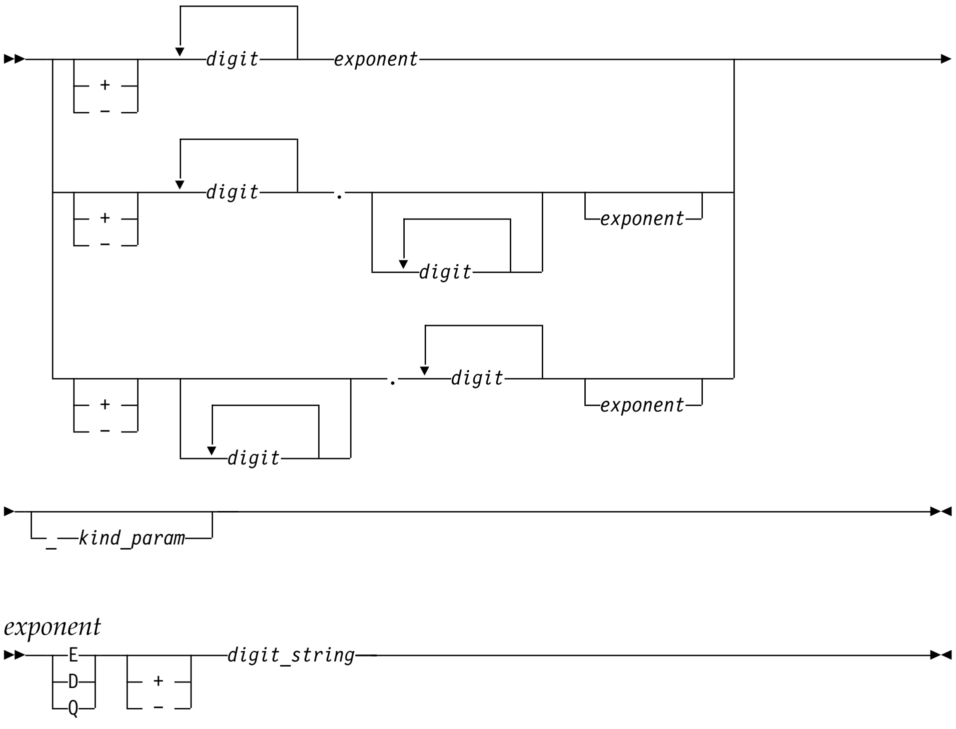
This ANSI/IEEE binary floating-point format also provides representations for +infinity, -infinity, and NaN (not-a-number) values. A NaN can be further classified as a quiet NaN (NaNQ) or a signaling NaN (NaNS). See "XL Fortran Floating-Point Processing" in the *User's Guide* for details on the internal representation of NaN values.

A real type specifier must include either the **REAL** keyword or the **DOUBLE PRECISION** keyword. The precision of **DOUBLE PRECISION** values is twice that of default real values. (The term *single precision* refers to the IEEE 4-byte representation, and the term *double precision* refers to the IEEE 8-byte representation.) See "REAL" on page 381 and "DOUBLE PRECISION" on page 290 for details on declaring entities of type real.

The forms of a real literal constant are:
- A basic real constant optionally followed by a kind type parameter
- A basic real constant followed by an exponent and an optional kind type parameter
- An integer constant (with no *kind_param*) followed by an exponent and an optional kind type parameter

A basic real constant has, in order, an optional sign, an integer part, a decimal point, and a fractional part. Both the integer part and fractional part are strings of digits; you can omit either of these parts, but not both. You can write a basic real constant with more digits than XL Fortran will use to approximate the value of the constant. XL Fortran interprets a basic real constant as a decimal number.

The form of a real constant is:



kind_param is either a *digit-string* or a *scalar-int-constant-name*

digit_string denotes a power of 10. **E** specifies a constant of type default real. **D** specifies a constant of type default **DOUBLE PRECISION**. **Q** specifies a constant of type **REAL(16)**.

If both *exponent* and *kind_param* are specified, the exponent letter must be **E**. If **D** or **Q** is specified, *kind_param* must not be specified.

A real literal constant that is specified without an exponent and a kind type parameter is of type default real.

Examples of Real Constants

```
+0.
+5.432E02_16   ! 543.2 in 16-byte representation
7.E3
3.4Q-301       ! Extended-precision constant
```

Complex

A complex type specifier must include either:
- the **COMPLEX** keyword, or
- the **DOUBLE COMPLEX** keyword

The following table shows the values that XL Fortran can represent for the kind type parameter and the length specification when the complex type specifier has the **COMPLEX** keyword:

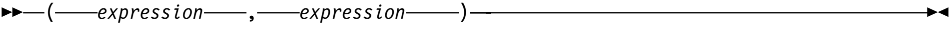
Kind Type Parameter *i* **COMPLEX(*i*)**	**Length Specification** *j* **COMPLEX*j**
4	8
8	16
16	32

The kind type parameter specifies the precision of each part of the complex entity, while the length specification specifies the length of the whole complex entity. The kind of a complex constant is determined by the kind of the constants in the real and imaginary parts.

The precision of **DOUBLE COMPLEX** values is twice that of default complex values.

See "COMPLEX" on page 270 and "DOUBLE COMPLEX" on page 287 for details on declaring entities of type complex.

Scalar values of type complex can be formed using complex constructors. The form of a complex constructor is:



A complex literal constant is a complex constructor where each expression is a pair of initialization expressions.

In Fortran 95 you are only allowed to use a single signed integer, or real literal constant in each part of the complex constructor. In addition, as an XL Fortran extension you can use variables and expressions in each part of the complex constructor.

If both parts of the literal constant are of type real, the kind type parameter of the literal constant is the kind parameter of the part with the greater precision, and the kind type parameter of the part with lower precision is converted to that of the other part.

If both parts are of type integer, they are each converted to type default real. If one part is of type integer and the other is of type real, the integer is converted to type real with the precision of type real.

Each part of a complex number has the following internal representation: a sign bit (s), a biased exponent (e), and a fraction (f).

```
COMPLEX(4) (equivalent to COMPLEX*8)
Bit no. 0....|....1....|....2....|....3....|....4....|....5....|....6...
        seeeeeeeefffffffffffffffffffffffseeeeeeeefffffffffffffffffffffff
COMPLEX(8) (equivalent to COMPLEX*16)
Bit no. 0....|....1....|....2....|....3....|....4....|....5....|....6...
        seeeeeeeeeeeffffffffffffffffffffffffffffffffffffffffffffffffffff
Bit no. .|....7....|....8....|....9....|....0....|....1....|....2....|..
        seeeeeeeeeeeffffffffffffffffffffffffffffffffffffffffffffffffffff
COMPLEX(16) (equivalent to COMPLEX*32)
Bit no. 0....|....1....|....2....|....3....|....4....|....5....|....6...
        seeeeeeeeeeeffffffffffffffffffffffffffffffffffffffffffffffffffff
Bit no. .|....7....|....8....|....9....|....0....|....1....|....2....|..
        seeeeeeeeeeeffffffffffffffffffffffffffffffffffffffffffffffffffff
Bit no. ...3....|....4....|....5....|....6....|....7....|....8....|....9
        seeeeeeeeeeeffffffffffffffffffffffffffffffffffffffffffffffffffff
Bit no. ....|....0....|....1....|....2....|....3....|....4....|....5....
        seeeeeeeeeeeffffffffffffffffffffffffffffffffffffffffffffffffffff
```

Examples of Complex Constants

```
(3_2,-1.86)         ! Integer constant 3 is converted to default real
                    ! for constant 3.0

(45Q6,6D45)         ! The imaginary part is converted to extended
                    ! precision 6.Q45

(1+1,2+2)           ! Use of constant expressions. Both parts are
                    ! converted to default real
```

Logical

The following table shows the values that XL Fortran can represent using the logical data type:

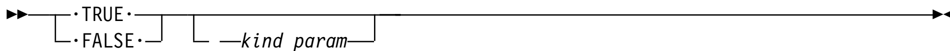
Kind parameter	Values	Internal (hex) Representation
1	.TRUE. .FALSE.	01 00
2	.TRUE. .FALSE.	0001 0000
4	.TRUE. .FALSE.	00000001 00000000
8	.TRUE. .FALSE.	0000000000000001 0000000000000000

Note: Any internal representation other than 1 for .TRUE. and 0 for .FALSE. is undefined.

XL Fortran sets the default kind type parameter to 4. The kind type parameter is equivalent to the byte size for logical values. Use the **-qintsize** compiler option to change the default logical size to 2, 4, or 8 bytes. Note that the **-qintsize** option similarly affects the default integer size.

The logical type specifier must include the **LOGICAL** keyword. See "LOGICAL" on page 347 for details on declaring entities of type logical.

The form of a logical literal constant is:



kind_param
: is either a *digit-string* or a *scalar-int-constant-name*

A logical constant can have a logical value of either true or false.

You can also use the abbreviations T and F (without the periods) for .TRUE. and .FALSE, respectively, but only in formatted input, or as initial values in **DATA** statements, **STATIC** statements, or type declaration statements. A kind type parameter cannot be specified for the abbreviated form. If T or F has been defined as a named constant, it is treated as a named constant rather than the logical literal constant.

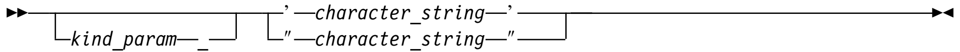
Examples of Logical Constants

```
.FALSE._4
.TRUE.
```

Character

The character type specifier must include the **CHARACTER** keyword. See "CHARACTER" on page 260 for details on declaring entities of type character.

The form of a character literal constant is:



kind_param
: is either a *digit-string* or a *scalar-int-constant-name*

XL Fortran supports a kind type parameter value of 1, representing the ASCII collating sequence.

Character literal constants can be delimited by double quotation marks as well as apostrophes.

character_string consists of any characters capable of representation in XL Fortran, except the new-line character (`\n`), because it is interpreted as the end of the source line. The delimiting apostrophes (′) or double quotation marks (") are not part of the data represented by the constant. Blanks embedded between these delimiters are significant.

If a string is delimited by apostrophes, you can represent an apostrophe within the string with two consecutive apostrophes (without intervening blanks). If a string is delimited by double quotation marks, you can represent a double quotation mark within the string with two consecutive double quotation marks (without intervening blanks). The two consecutive apostrophes or double quotation marks will be treated as one character.

You can place a double quotation mark within a character literal constant delimited by apostrophes to represent a double quotation mark, and an apostrophe character within a character constant delimited by double quotation marks to represent a single apostrophe.

The length of a character literal constant is the number of characters between the delimiters, except that each pair of consecutive apostrophes or double quotation marks counts as one character.

Each character object requires 1 byte of storage.

A zero-length character object uses no storage.

For compatibility with C language usage, XL Fortran recognizes the following escape sequences in character strings:

Escape	Meaning
\b	Backspace
\f	Form feed
\n	New-line
\t	Tab
\0	Null
\′	Apostrophe (does not terminate a string)

Escape	Meaning
\″	Double quotation mark (does not terminate a string)
\\	Backslash
\x	x, where x is any other character

To ensure that scalar character initialization expressions in procedure references are terminated with null characters (\0) for C compatibility, use the **-qnullterm** compiler option (see ″-qnullterm Option″ in the *User's Guide* for details and exceptions).

All escape sequences represent a single character.

If you do not want these escape sequences treated as a single character, specify the **-qnoescape** compiler option (see ″-qescape Option″ in the *XL Fortran for AIX User's Guide*). The backslash will have no special significance.

The maximum length of a character literal constant depends on the maximum number of characters allowed in a statement.

If you specify the **-qctyplss** compiler option, character constant expressions are treated as if they are Hollerith constants. See "Hollerith Constants" on page 48 for information on Hollerith constants. For information on the **-qctyplss** compiler option, see ″-qctyplss Option″ in the *User's Guide*

XL Fortran supports multibyte characters within character literal constants, Hollerith constants, **H** edit descriptors, character-string edit descriptors, and comments through the **-qmbcs** compiler option.

Support is also provided for Unicode characters and filenames. If the environment variable **LANG** is set to **UNIVERSAL** and the **-qmbcs** compiler option is specified, the compiler can read and write Unicode characters and filenames. (See the *User's Guide* for more information.)

Examples of Character Constants

```
''                    ! Zero-length character constant
1_"ABCDEFGHIJ"        ! Character constant of length 10, with kind 1
'\"\2\'\A567\\\\\''   ! Character constant of length 10 "2'A567\\'
```

Character Substrings

A character substring is a contiguous portion of a character string (called a parent string), which is a scalar variable name, scalar constant, scalar structure component, or array element. A character substring is identified by a substring reference whose form is:

```
►►──┬─scalar_variable_name─┬──(──┬───────────┬──:──┬───────────┬──)──────────►◄
    ├─array_element────────┤     └─int_expr1─┘     └─int_expr2─┘
    ├─scalar_constant──────┤
    └─scalar_struct_comp───┘
```

int_expr1 and *int_expr2*
: specify the leftmost character position and rightmost character position, respectively, of the substring. Each is a scalar integer expression called a substring expression.

The length of a character substring is the result of the evaluation of `MAX(int_expr2 - int_expr1 + 1,0)`.

If *int_expr1* is less than or equal to *int_expr2*, their values must be such that:
    Rule One: 1 ≤ *int_expr1* ≤ *int_expr2* ≤ *length*

where *length* is the length of the parent string. If *int_expr1* is omitted, its default value is 1. If *int_expr2* is omitted, its default value is *length*.

Previous versions of XL Fortran adhere to FORTRAN 77 constraints on substring expressions, as noted in Rule One above. To perform compile-time checking on substring bounds in accordance with FORTRAN 77 rules, use the **-qnozerosize** compiler option. For Fortran 90 compliance, use **-qzerosize**. To perform run-time checking on substring bounds, use both the **-qcheck** option and the **-qzerosize** (or **-qnozerosize**) option. (See the *User's Guide* for more information.)

A substring of an array section is treated differently. See "Array Sections and Substring Ranges" on page 78.

Examples of Character Substrings:

```
CHARACTER(8) ABC, X, Y, Z
ABC = 'ABCDEFGHIJKL'(1:8)   ! Substring of a constant
X = ABC(3:5)                ! X = 'CDE'
Y = ABC(-1:6)               ! Not allowed in either FORTRAN 77 or Fortran 90
Z = ABC(6:-1)               ! Z = ' valid only in Fortran 90
```

BYTE

The byte type specifier is the **BYTE** keyword. See "BYTE" on page 253 for details on declaring entities of type byte.

The **BYTE** intrinsic data type does not have its own literal constant form. A **BYTE** data object is treated as an **INTEGER(1)**, **LOGICAL(1)**, or **CHARACTER(1)** data object, depending on how it is used. See "Using Typeless Constants" on page 49.

Derived Types

You can create additional data types, known as derived types, from intrinsic data types and other derived types. You require a type definition to define the name of the derived type (*type_name*), as well as the data types and names of the components of the derived type.

DERIVED_TYPE_statement
: See "Derived Type" on page 282 for syntax details.

PRIVATE_SEQUENCE_block
: includes the **PRIVATE** statement (keyword only) and/or the **SEQUENCE** statement. Only one of each statement can be specified. See "PRIVATE" on page 370 and "SEQUENCE" on page 391 for details on syntax.

component_def_stmt_block
: consists of one or more type declaration statements to define the components of the derived type. The type declaration statements can specify only the **DIMENSION** and **POINTER** attributes. See "Type Declaration" on page 402 for detailed syntax and information.

  In addition, Fortran 95 allows you to specify a default initialization for each component in the definition of a derived type. See "Type Declaration" on page 402 for detailed syntax and information.

END_TYPE_statement
: See "END TYPE" on page 301.

Direct components of a derived type are:

- the components of that type
- the direct components of a derived type component without **POINTER** attribute.

Each derived type is resolved into ultimate components of intrinsic data type.

The type name is a local entity. It cannot be the same name as any of the intrinsic data types except **BYTE** and **DOUBLE COMPLEX**.

The **END TYPE** statement can optionally contain the same *type_name* as specified on the **TYPE** statement.

The components of a derived type can specify any of the intrinsic data types. Components can also be of a previously defined derived type. A pointer component can be of the same derived type that it is a component of. Within a derived type, the names of components must be unique, although they can be different from names outside the scope of the derived-type definition. Components that are declared to be of type **CHARACTER** must have length specifications that are constant specification expressions; asterisks are not allowed as length specifiers. Nonpointer array components must be declared with constant dimension declarators. Pointer array components must be declared with a *deferred_shape_spec_list*.

By default, no storage sequence is implied by the order of the component definitions. However, if you specify the **SEQUENCE** statement, the derived type becomes a *sequence derived type*. For a sequence derived type, the order of the components specifies a storage sequence for objects declared with this derived type. If a component of a sequence derived type is of a derived type, that derived type must also be a sequence derived type.

The size of a sequence derived type is equal to the number of bytes of storage needed to hold all of the components of that derived type.

Use of sequence derived types can lead to misaligned data, which can adversely affect the performance of the program.

The **PRIVATE** statement can only be specified if the derived-type definition is within the specification part of a module. If a component of a derived type is of a type declared to be private, either the derived-type definition must contain the **PRIVATE** statement or the derived type itself must be private.

If a type definition is private, the following are accessible only within the defining module:
- The type name
- Structure constructors for the type
- Any entity of the type
- Any procedure that has a dummy argument or function result of the type

If a derived-type definition contains a **PRIVATE** statement, its components are accessible only within the defining module, even if the derived type itself is public. Structure components can only be used in the defining module.

A component of a derived-type entity cannot appear as an input/output list item if any ultimate component of the object cannot be accessed by the scoping unit of the input/output statement. A derived-type object cannot appear in a data transfer statement if it has a component that is a pointer.

A scalar entity of derived type is called a *structure*. A scalar entity of sequence derived type is called a *sequence structure*. The type specifier of a structure must include the **TYPE** keyword, followed by the name of the derived type in parentheses. See the **TYPE** type declaration statement on page 399 for details on declaring entities of a specified derived type. The components of a structure are called *structure components*. A *structure component* is one of the components of a structure or is an array whose elements are components of the elements of an array of derived type.

An object of a private derived type cannot be used outside the defining module.

A candidate data object for default initialization is a named data object that:
1. is of derived type with default initialization specified for any of its direct components.
2. has neither the **POINTER**, nor the **ALLOCATABLE** attribute.
3. is not use or host associated.
4. is not a pointee.

A default initialization for a non-pointer component will take precedence over any default initialization appearing for any direct component of its type.

If a dummy argument with **INTENT(OUT)** is of a derived type with default initialization, it must not be an assumed-size array. If a non-pointer object or subobject has been specified with default initialization in a type definition, it must not be initialized by a **DATA** statement. You must not specify data objects of derived type with default initializations in a common block.

You can specify a default initialization for some components of a derived type, but it is not necessary for every component.

You can specify default initialization for a storage unit that is storage associated. However, the objects or subobjects supplying the default initialization must be of the same type. The objects or subobjects must also have the same type parameters and supply the same value for the storage unit.

Unlike explicit initialization, it is not necessary for a data object to have the **SAVE** attribute for component default initialization to have an impact. In addition, default initialization does not imply the **SAVE** attribute.

A direct component will receive an initial value if you specify a default initialization on the corresponding component definition in the type definition, regardless of the accessibility of the component.

For candidate data objects for default initialization, their nonpointer direct components are either initially defined, or become defined by their corresponding default initialization expressions, and their pointer direct components are either initially disassociated, or become disassociated if one of the following conditions is met:

- become initially defined or disassociated:
  - the data object in question has the **SAVE** attribute.
  - if you declare the data object in question in a **BLOCK DATA** unit, module, or main program unit.
- become defined or disassociated:
  - a function with the data object in question as its result is invoked
  - a procedure with the data object in question as an **INTENT(OUT)** dummy argument is invoked.
  - a procedure with the data object in question as a local object is invoked, and the data object does not have the **SAVE** attribute.

Allocation of an object of a derived type in which you specify a default initialization for a direct component will cause the component to:

- become defined, if it is a non-pointer direct component
- become disassociated, if it is a pointer direct component

In a subprogram with an **ENTRY** statement, default initialization only occurs for the dummy arguments that appear in the argument list of the procedure name referenced. If such a dummy argument has the **OPTIONAL** attribute, default initialization will only occur if the dummy argument is present.

Module data objects, which are of derived type with default initializations must have the **SAVE** attribute, if they are candidate data objects for default initialization.

Determining Type for Derived Types

Two data objects have the same derived type if they are declared with reference to the same derived-type definition.

If the data objects are in different scoping units, they can still have the same derived type. Either the derived-type definition is accessible via host or use association, or the data objects reference their own derived-type definitions with the following conditions:

- The derived-type definitions have the same name. Renaming affects only the local name, not the original definition of the derived type.

- Each of the derived-type definitions contains the **SEQUENCE** statement.
- Each derived-type definition has components that do not specify private accessibility and that match the components of the other derived type in name, order, and attributes.

A derived-type definition that specifies **SEQUENCE** is not the same as a definition declared to be private or that has components that are private.

Example of Determining Type with Derived Types

```
PROGRAM MYPROG

TYPE NAME                          ! Sequence derived type
   SEQUENCE
   CHARACTER(20) LASTNAME
   CHARACTER(10) FIRSTNAME
   CHARACTER(1)  INITIAL
END TYPE NAME
TYPE (NAME) PER1

CALL MYSUB(PER1)
PER1 = NAME('Smith','John','K')    ! Structure constructor
CALL MYPRINT(PER1)

CONTAINS
  SUBROUTINE MYSUB(STUDENT)        ! Internal subroutine MYSUB
     TYPE (NAME) STUDENT           ! NAME is accessible via host association
   ⋮

  END SUBROUTINE MYSUB
END

SUBROUTINE MYPRINT(NAMES)          ! External subroutine MYPRINT
  TYPE NAME                        ! Same type as data type in MYPROG
     SEQUENCE
     CHARACTER(20) LASTNAME
     CHARACTER(10) FIRSTNAME
     CHARACTER(1)  INITIAL
  END TYPE NAME
  TYPE (NAME) NAMES                ! NAMES and PER1 from MYPROG
  PRINT *, NAMES                   ! have the same data type
END SUBROUTINE
```

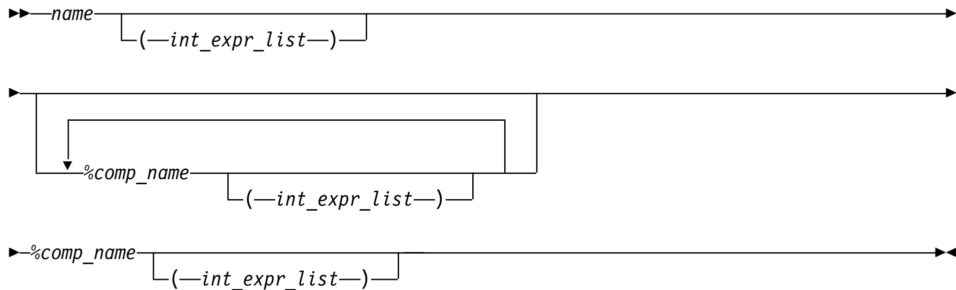
Structure Components

Structure components can be of any explicit type, including derived type.

Note: The case in which a structure component has a subobject that is an array or array section requires some background information from “Array Sections” on page 74, and is explained in “Array Sections and

Structure Components” on page 79. The following rules for scalar structure components apply also to structure components that have array subobjects.

You can refer to a specific structure component using a *component designator*. A scalar component designator has the following syntax:



name is the name of an object of derived type

comp_name is the name of a derived-type component

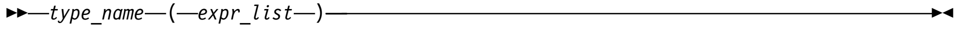
int_expr is a scalar integer or real expression called a subscript expression

The structure component has the same type, type parameters, and **POINTER** attribute (if any) as the right-most *comp_name*. It inherits any **INTENT**, **TARGET**, and **PARAMETER** attributes from the parent object.

Notes:

1. Each *comp_name* must be a component of the immediately preceding *name* or *comp_name*.
2. The *name* and each *comp_name*, except the right-most, must be of derived type.
3. The number of subscript expressions in any *int_expr_list* must equal the rank of the preceding *name* or *comp_name*.
4. If *name* or any *comp_name* is the name of an array, it must have an *int_expr_list*.
5. The right-most *comp_name* must be scalar.

Structure Constructor



type_name is the name of the derived type

expr is an expression. Expressions are defined under “Chapter 5. Expressions and Assignment” on page 85.

A structure constructor allows a scalar value of derived type to be constructed from an ordered list of values. A structure constructor must not appear before the definition of the referenced derived type.

expr_list contains one value for each component of the derived type. The sequence of expressions in the *expr_list* must agree in number and order with the components of the derived type. The type and type parameters of each expression must be assignment-compatible with the type and type parameters of the corresponding component. Data type conversion is performed if necessary.

A component that is a pointer can be declared with the same type that it is a component of. If a structure constructor is created for a derived type containing a pointer, the expression corresponding to the pointer component must evaluate to an object that would be an allowable target for such a pointer in a pointer assignment statement.

Examples of Derived Types:

Example 1:

```
MODULE PEOPLE
  TYPE NAME
     SEQUENCE                    ! Sequence derived type
     CHARACTER(20) LASTNAME
     CHARACTER(10) FIRSTNAME
     CHARACTER(1)  INITIAL
  END TYPE NAME

  TYPE PERSON                    ! Components accessible via use
                                 ! association
     INTEGER AGE
     INTEGER BIRTHDATE(3)        ! Array component
     TYPE (NAME) FULLNAME        ! Component of derived type
  END TYPE PERSON
END MODULE PEOPLE
PROGRAM TEST1
  USE PEOPLE
  TYPE (PERSON) SMITH, JONES
  SMITH = PERSON(30, (/6,30,63/), NAME('Smith','John','K'))
                                 ! Nested structure constructors
  JONES%AGE = SMITH%AGE          ! Component designator
  CALL TEST2
  CONTAINS

  SUBROUTINE TEST2
    TYPE T
      INTEGER EMP_NO
```

```
      CHARACTER, POINTER :: EMP_NAME(:)  ! Pointer component
    END TYPE T
    TYPE (T) EMP_REC
    CHARACTER, TARGET :: NAME(10)
    EMP_REC = T(24744,NAME)              ! Pointer assignment occurs
  END SUBROUTINE                         ! for EMP_REC%EMP_NAME
END PROGRAM
```

Example 2:

```
PROGRAM LOCAL_VAR
   TYPE DT
      INTEGER A
      INTEGER :: B = 80
   END TYPE

   TYPE(DT) DT_VAR                      ! DT_VAR%B IS INITIALIZED
END PROGRAM LOCAL_VAR
```

Example 3:

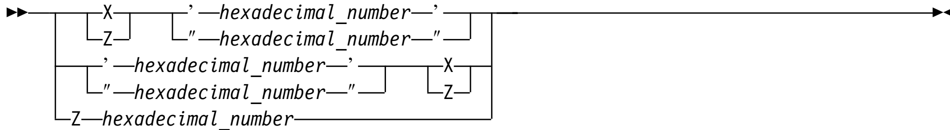
```
MODULE MYMOD
   TYPE DT
      INTEGER :: A = 40
      INTEGER, POINTER :: B => NULL()
   END TYPE
END MODULE

PROGRAM DT_INIT
   USE MYMOD
   TYPE(DT), SAVE :: SAVED(8)            ! SAVED%A AND SAVED%B ARE INITIALIZED
   TYPE(DT) LOCAL(5)                     ! LOCAL%A LOCAL%B ARE INITIALIZED
END PROGRAM
```

Typeless Literal Constants

A typeless constant does not have an intrinsic type. Hexadecimal, octal, binary, and Hollerith constants can be used in any situation where intrinsic literal constants are used, except as the length specification in a type declaration statement (although typeless constants can be used in a *type_param_value* in **CHARACTER** type declaration statements). The number of digits recognized in a hexadecimal, octal, or binary constant depends on the context in which the constant is used.

Hexadecimal Constants

The form of a hexadecimal constant is:



hexadecimal_number
is a string composed of digits (0-9) and letters (A-F, a-f). Corresponding uppercase and lowercase letters are equivalent.

The **Z***nn...nn* form of a hexadecimal constant can only be used as a data initialization value delimited by slashes. If this form of a hexadecimal constant is the same string as the name of a constant you defined previously with the **PARAMETER** attribute, XL Fortran recognizes the string as the named constant.

If 2x hexadecimal digits are present, x bytes are represented.

See "Using Typeless Constants" on page 49 for information on how XL Fortran interprets the constant.

Examples of Hexadecimal Constants

```
Z'0123456789ABCDEF'
Z"FEDCBA9876543210
Z'0123456789aBcDeF'
Z0123456789aBcDeF   ! This form can only be used as an initialization value
```

Octal Constants

The form of an octal constant is:

octal_number
is a string composed of digits (0-7)

Because an octal digit represents 3 bits, and a data object represents a multiple of 8 bits, the octal constant may contain more bits than are needed by the data object. For example, an **INTEGER(2)** data object can be represented by a 6-digit octal constant if the leftmost digit is 0 or 1; an **INTEGER(4)** data object can be represented by an 11-digit constant if the leftmost digit is 0, 1, 2, or 3; an **INTEGER(8)** can be represented by a 22-digit constant if the leftmost digit is 0 or 1.

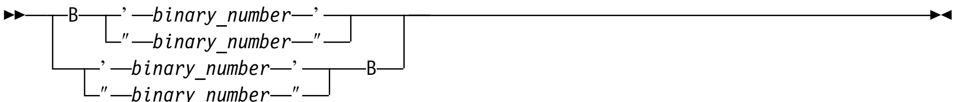
See “Using Typeless Constants” on page 49 for information on how the constant is interpreted by XL Fortran.

Examples of Octal Constants

```
O'01234567'
"01234567"O
```

Binary Constants

The form of a binary constant is:



binary_number is a string formed from the digits 0 and 1

If 8x binary digits are present, x bytes are represented.

See “Using Typeless Constants” on page 49 for information on how XL Fortran interprets the constant.

Examples of Binary Constants

```
B"10101010"
'10101010'B
```

Hollerith Constants

The form of a Hollerith constant is:

```
>>—n—H—character_string——————————————————————————————————><
```

A Hollerith constant consists of a nonempty string of characters capable of representation in the processor and preceded by nH, where n is a positive unsigned integer constant representing the number of characters after the H. n cannot specify a kind type parameter. The number of characters in the string may be from 1 to 255.

Note: If you specify nH and fewer than n characters are specified after the n, any blanks that are used to extend the input line to the right margin are considered to be part of the Hollerith constant. A Hollerith constant can be continued on a continuation line. At least n characters must be available for the Hollerith constant.

XL Fortran also recognizes escape sequences in Hollerith constants, unless the **-qnoescape** compiler option is specified. If a Hollerith constant contains an escape sequence, n is the number of characters in the internal representation

of the string, not the number of characters in the source string. (For example, `2H\"\"` represents a Hollerith constant for two double quotation marks.)

XL Fortran provides support for multibyte characters within character constants, Hollerith constants, **H** edit descriptors, character-string edit descriptors, and comments. This support is provided through the **-qmbcs** option. Assignment of a constant containing multibyte characters to a variable that is not large enough to hold the entire string may result in truncation within a multibyte character.

Support is also provided for Unicode characters and filenames. If the environment variable **LANG** is set to **UNIVERSAL** and the **-qmbcs** compiler option is specified, the compiler can read and write Unicode characters and filenames.

See "Using Typeless Constants" below for information on how XL Fortran interprets the constant.

Using Typeless Constants

The data type and length of a typeless constant are determined by the context in which you use the typeless constant. XL Fortran does not convert them before use.

- If you compile your program with the **-qctyplss** compiler option, character initialization expressions follow the rules that apply to Hollerith constants.
- A typeless constant can assume only one of the intrinsic data types.
- When you use a typeless constant with an arithmetic or logical unary operator, the constant assumes a default integer type.
- When you use a typeless constant with an arithmetic, logical, or relational binary operator, the constant assumes the same data type as the other operand. If both operands are typeless constants, they assume a type of default integer unless both operands of a relational operator are Hollerith constants. In this case, they both assume a character data type.
- When you use a typeless constant in a concatenation operation, the constant assumes a character data type.
- When you use a typeless constant as the expression on the right-hand side of an assignment statement, the constant assumes the type of the variable on the left-hand side.
- When you use a typeless constant in a context that requires a specific data type, the constant assumes that data type.
- When you use a typeless constant as an initial value in a **DATA** statement, **STATIC** statement, or type declaration statement, or as the constant value of a named constant in a **PARAMETER** statement, or when the typeless constant is to be treated as any noncharacter type of data, the following rules apply:

- If a hexadecimal, octal, or binary constant is smaller than the length expected, XL Fortran adds zeros on the left. If it is longer, the compiler truncates on the left.
- If a Hollerith constant is smaller than the length expected, the compiler adds blanks on the right. If it is longer, the compiler truncates on the right.
- If a typeless constant specifies the value of a named constant with a character data type having inherited length, the named constant has a length equal to the number of bytes specified by the typeless constant.

- When a typeless constant is treated as an object of type character (except when used as an initial value in a **DATA**, **STATIC**, type declaration, or component definition statement).
- When you use a typeless constant as part of a complex constant, the constant assumes the data type of the other part of the complex constant. If both parts are typeless constants, the constants assume the real data type with length sufficient to represent both typeless constants.
- When you use a typeless constant as an actual argument, the type of the corresponding dummy argument must be an intrinsic data type. The dummy argument must not be a procedure, pointer, array, object of derived type, or alternate return specifier.
- When you use a typeless constant as an actual argument, and:
  - The procedure reference is to a generic intrinsic procedure,
  - All of the arguments are typeless constants, and
  - There *is* a specific intrinsic procedure that has the same name as the generic procedure name,

  the reference to the generic name will be resolved through the specific procedure.
- When you use a typeless constant as an actual argument, and:
  - The procedure reference is to a generic intrinsic procedure,
  - All of the arguments are typeless constants, and
  - There is *no* specific intrinsic procedure that has the same name as the generic procedure name,

  the typeless constant is converted to default integer. If a specific intrinsic function takes integer arguments, the reference is resolved through that specific function. If there are no specific intrinsic functions, the reference is resolved through the generic function.
- When you use a typeless constant as an actual argument, and:
  - The procedure reference is to a generic intrinsic procedure, and
  - There is another argument specified that is not a typeless constant,

  the typeless constant assumes the type of that argument. The selected specific intrinsic procedure is based on that type.
- When you use a typeless constant as an actual argument, and the procedure name is established to be generic but is not an intrinsic procedure, the

generic procedure reference must resolve to only one specific procedure. The constant assumes the data type of the corresponding dummy argument of that specific procedure. For example:

```
INTERFACE SUB
  SUBROUTINE SUB1( A )
     REAL A
  END SUBROUTINE
  SUBROUTINE SUB2( A, B )
     REAL A, B
  END SUBROUTINE
  SUBROUTINE SUB3( I )
     INTEGER I
  END SUBROUTINE
END INTERFACE
CALL SUB('C0600000'X, '40066666'X)  ! Resolves to SUB2

CALL SUB('00000000'X)               ! Invalid - ambiguous, may
                                    ! resolve to either SUB1 or SUB3
```

- When you use a typeless constant as an actual argument, and the procedure name is established to be only specific, the constant assumes the data type of the corresponding dummy argument.
- When you use a typeless constant as an actual argument, and:
  - The procedure name has not been established to be either generic or specific, and
  - The constant has been passed by reference,

  the constant assumes the default integer size but no data type, unless it is a Hollerith constant. The default for passing a Hollerith constant is the same as if it were a character actual argument. See “Resolution of Procedure References” on page 172 for more information about establishing a procedure name to be generic or specific.
- When you use a typeless constant as an actual argument, and:
  - The procedure name has not been established to be either generic or specific, and
  - The constant has been passed by value,

  the constant is passed as if it were a default integer for hexadecimal, binary, and octal constants.

  If the constant is a Hollerith constant and it is smaller than the size of a default integer, XL Fortran adds blanks on the right. If the constant is a Hollerith constant and it is larger than 8 bytes, XL Fortran truncates the rightmost Hollerith characters. See “Resolution of Procedure References” on page 172 for more information about establishing a procedure name to be generic or specific.

- When you use a typeless constant in any other context, the constant assumes the default integer type, with the exception of Hollerith constants. Hollerith constants assume a character data type when used in the following situations:
  - An H edit descriptor
  - A relational operation with both operands being Hollerith constants
  - An input/output list
- If a typeless constant is to be treated as a default integer but the value cannot be represented within the value range for a default integer, the constant is promoted to a representable kind.

Examples of Typeless Constants in Expressions

```
INT=B'1'          ! Binary constant is default integer
RL4=X'1'          ! Hexadecimal constant is default real
INT=INT + O'1'    ! Octal constant is default integer
RL4=INT + B'1'    ! Binary constant is default integer
INT=RL4 + Z'1'    ! Hexadecimal constant is default real
ARRAY(O'1')=1.0   ! Octal constant is default integer

LOGICAL(8) LOG8
LOG8=B'1'         ! Binary constant is LOGICAL(8), LOG8 is .TRUE.
```

How Type Is Determined

Each user-defined function or named entity has a data type. (The type of an entity accessed by host or use association is determined in the host scoping unit or accessed module, respectively.) The type of a name is determined, in the following sequence, in one of three ways:

1. Explicitly, in one of the following ways:
   - From a specified type declaration statement (see "Type Declaration" on page 402 for details).
   - For function results, from a specified type statement or its **FUNCTION** statement.
2. Implicitly, from a specified **IMPLICIT** type statement (see "IMPLICIT" on page 329 for details).
3. Implicitly, by predefined convention. By default (that is, in the absence of an **IMPLICIT** type statement), if the first letter of the name is `I`, `J`, `K`, `L`, `M`, or `N`, the type is default integer. Otherwise, the type is default real.

In a given scoping unit, if a letter, dollar sign, or underscore has not been specified in an **IMPLICIT** statement, the implicit type used is the same as the implicit type used by the host scoping unit. A program unit and interface body are treated as if they had a host with an **IMPLICIT** statement listing the predefined conventions.

The data type of a literal constant is determined by its form.

Definition Status of Variables

A variable is always defined or undefined, and its definition status can change during program execution. A named constant has a value and cannot be defined or redefined during program execution.

Arrays (including sections), structures, and variables of character or complex type are objects made up of zero or more subobjects. Associations can be established between variables and subobjects and between subobjects of different variables.

- An object is defined if all of its subobjects are defined. That is, each object or subobject has a value that does not change until it becomes undefined or until it is redefined with a different value.
- If an object is undefined, at least one of its subobjects is undefined. An undefined object or subobject cannot provide a predictable value.

Variables are initially defined if they are specified to have initial values by **DATA** statements, type declaration statements, or **STATIC** statements. In addition, default initialization may cause a variable to be initially defined. Zero-sized arrays and zero-length character objects are always defined.

All other variables are initially undefined.

Events Causing Definition

The following events will cause a variable to become defined:

1. Execution of an intrinsic assignment statement other than a masked array assignment statement or **FORALL** assignment statement causes the variable that precedes the equal sign to become defined.

   Execution of a defined assignment statement may cause all or part of the variable that precedes the equal sign to become defined.
2. Execution of a masked array assignment or **FORALL** assignment statement may cause some or all of the array elements in the assignment statement to become defined.
3. As execution of an input statement proceeds, each variable that is assigned a value from the input file becomes defined at the time that data are transferred to it. Execution of a **WRITE** statement whose unit specifier identifies an internal file causes each record that is written to become defined.

   As execution of an asynchronous input statement proceeds, the variable does not become defined until the matching **WAIT** statement is executed.
4. Execution of a **DO** statement causes the **DO** variable, if any, to become defined.
5. Default initialization may cause a variable to be initially defined.

6. Beginning of execution of the action specified by an implied-**DO** list in an input/output statement causes the implied-**DO** variable to become defined.
7. Execution of an **ASSIGN** statement causes the variable in the statement to become defined with a statement label value.
8. A reference to a procedure causes the entire dummy argument data object to become defined if the entire corresponding actual argument is defined with a value that is not a statement label.

   A reference to a procedure causes a subobject of a dummy argument to become defined if the corresponding subobject of the corresponding actual argument is defined.
9. Execution of an input/output statement containing an **IOSTAT=** specifier causes the specified integer variable to become defined.
10. Execution of a **READ** or **WRITE** statement containing an **ID=** specifier causes the specified integer variable to become defined.
11. Execution of a **WAIT** statement containing a **DONE=** specifier causes the specified logical variable to become defined.
12. Execution of a **READ** statement containing a **SIZE=** specifier causes the specified integer variable to become defined.
13. Execution of a synchronous **READ** or **WRITE** statement containing a **NUM=** specifier causes the specified integer variable to become defined.

    Execution of an asynchronous **READ** or **WRITE** statement containing a **NUM=** specifier does not cause the specified integer variable to become defined. The integer variable is defined upon execution of the matching **WAIT** statement.
14. Execution of an **INQUIRE** statement causes any variable that is assigned a value during the execution of the statement to become defined if no error condition exists.
15. When a character storage unit becomes defined, all associated character storage units become defined.

    When a numeric storage unit becomes defined, all associated numeric storage units of the same type become defined, except that variables associated with the variable in an **ASSIGN** statement become undefined when the **ASSIGN** statement is executed. When an entity of type **DOUBLE PRECISION** becomes defined, all totally associated entities of double precision real type become defined.

    A nonpointer scalar object of type nondefault integer, real other than default or double precision, nondefault logical, nondefault complex, nondefault character of any length, or nonsequence type occupies a single unspecified storage unit that is different for each case. A pointer that is distinct from other pointers in at least one of type, kind, and rank

occupies a single unspecified storage unit. When an unspecified storage unit becomes defined, all associated unspecified storage units become defined.

When a default complex entity becomes defined, all partially associated default real entities become defined.

16. When both parts of a default complex entity become defined as a result of partially associated default real or default complex entities becoming defined, the default complex entity becomes defined.
17. When all components of a numeric sequence structure or character sequence structure become defined as a result of partially associated objects becoming defined, the structure becomes defined.
18. Execution of an **ALLOCATE** or **DEALLOCATE** statement with a **STAT=** specifier causes the variable specified by the **STAT=** specifier to become defined.
19. Allocation of a zero-sized array causes the array to become defined.
20. Invocation of a procedure causes any automatic object of zero size in that procedure to become defined.
21. Execution of a pointer assignment statement that associates a pointer with a target that is defined causes the pointer to become defined.
22. In a **FORALL** statement or construct, the *index-name* becomes defined when the *index-name* value set is evaluated.
23. If a variable in a **THREADPRIVATE** common block is explicitly initialized, each new thread's copy of the variable is initialized to the initial value of the variable when the thread is first created, unless it is the master thread.
24. If a **THREADPRIVATE** common block is specified in a **COPYIN** clause, each new thread duplicates the master thread's definition and association status of the variables in the common block. Therefore, if the master thread's copy of a variable is defined on entry to a parallel region, each new thread's copy of the variable will also be defined.
25. When a variable is specified in a **FIRSTPRIVATE** clause of a **PARALLEL**, **PARALLEL DO**, **DO**, **PARALLEL SECTIONS**, **SECTIONS** or **SINGLE** directive, each new thread duplicates the master thread's definition and association status of the variable. Therefore, if the master thread's copy of a variable is defined on entry to a parallel region, each new thread's copy of the variable will also be defined.

Events Causing Undefinition

The following events will cause a variable to become undefined:

1. When a variable of a given type becomes defined, all associated variables of different type become undefined. However, when a variable of type default real is partially associated with a variable of type default complex, the complex variable does not become undefined when the real

variable becomes defined and the real variable does not become undefined when the complex variable becomes defined. When a variable of type default complex is partially associated with another variable of type default complex, definition of one does not cause the other to become undefined.

2. Execution of an **ASSIGN** statement causes the variable in the statement to become undefined as an integer. Variables that are associated with the variable also become undefined.
3. If the evaluation of a function may cause an argument of the function or a variable in a module or in a common block to become defined, and if a reference to the function appears in an expression in which the value of the function is not needed to determine the value of the expression, the argument or variable becomes undefined when the expression is evaluated.
4. The execution of a **RETURN** statement or **END** statement within a subprogram causes all variables that are local to its scoping unit, or that are local to the current instance of its scoping unit for a recursive invocation, to become undefined, except for the following:
   a. Variables with the **SAVE** or **STATIC** attribute.
   b. Variables in blank common.
   c. According to Fortran 90, variables in a named common block that appears in the subprogram and appears in at least one other scoping unit that is making either a direct or indirect reference to the subprogram.

      XL Fortran does not undefine these variables, unless they are part of a threadlocal common block.
   d. Variables accessed from the host scoping unit.
   e. According to Fortran 90, variables accessed from a module that also is referenced directly or indirectly by at least one other scoping unit that is making either a direct or indirect reference to the subprogram.

      XL Fortran does not undefine these variables.
   f. According to Fortran 90, variables in a named common block that are initially defined and that have not been subsequently defined or redefined.

      XL Fortran does not undefine these variables.
5. When an error condition or end-of-file condition occurs during execution of an input statement, all of the variables specified by the input list or namelist-group of the statement become undefined.
6. When an error condition, end-of-file condition, or end-of-record condition occurs during execution of an input/output statement and the statement contains any implied-**DO** lists, all of the implied-**DO** variables in the statement become undefined.

7. Execution of a defined assignment statement may leave all or part of the variable that precedes the equal sign undefined.
8. Execution of a direct access input statement that specifies a record that has not been written previously causes all of the variables specified by the input list of the statement to become undefined.
9. Execution of an **INQUIRE** statement may cause the **NAME=**, **RECL=**, and **NEXTREC=** variables to become undefined.
10. When a character storage unit becomes undefined, all associated character storage units become undefined.

    When a numeric storage unit becomes undefined, all associated numeric storage units become undefined unless the undefinition is a result of defining an associated numeric storage unit of different type (see (1) above).

    When an entity of double precision real type becomes undefined, all totally associated entities of double precision real type become undefined.

    When an unspecified storage unit becomes undefined, all associated unspecified storage units become undefined.
11. A reference to a procedure causes part of a dummy argument to become undefined if the corresponding part of the actual argument is defined with a value that is a statement label value.
12. When an allocatable array is deallocated, it becomes undefined. Successful execution of an **ALLOCATE** statement causes the allocated array to become undefined.
13. Execution of an **INQUIRE** statement causes all inquiry specifier variables to become undefined if an error condition exists, except for the variable in the **IOSTAT=** specifier, if any.
14. When a procedure is invoked:
    a. An optional dummy argument that is not associated with an actual argument is undefined.
    b. A dummy argument with **INTENT(OUT)** is undefined.
    c. An actual argument associated with a dummy argument with **INTENT(OUT)** becomes undefined.
    d. A subobject of a dummy argument is undefined if the corresponding subobject of the actual argument is undefined.
    e. The function result variable is undefined, unless it was declared with the **STATIC** attribute and was defined in a previous invocation.
15. When the association status of a pointer becomes undefined or disassociated, the pointer becomes undefined.
16. When the execution of a **FORALL** statement or construct has completed, the *index-name* becomes undefined.

17. When a variable is specified in either the **PRIVATE** or **LASTPRIVATE** clause of a **PARALLEL**, **PARALLEL DO**, **DO**, **PARALLEL SECTIONS**, **SECTIONS** or **SINGLE** directive, each new thread's copy of the variable is undefined when the thread is first created.
18. When a variable is specified in a **FIRSTPRIVATE** clause of a **PARALLEL**, **PARALLEL DO**, **DO**, **PARALLEL SECTIONS**, **SECTIONS** or **SINGLE** directive, each new thread duplicates the master thread's definition and association status of the variable. Therefore, if the master thread's copy of a variable is undefined on entry to a parallel region, each new thread's copy of the variable will also be undefined.
19. When a variable is specified in the **NEW** clause of an **INDEPENDENT** directive, the variable is undefined at the beginning of every iteration of the following **DO** loop.
20. When a variable appears in asynchronous input, that variable becomes undefined, and remains undefined, until the matching **WAIT** statement is reached.
21. If a variable in a **THREADPRIVATE** common block is explicitly initialized, each new thread's copy of the variable is initialized to the initial value of the variable when the thread is first created, unless it is the master thread. If the master thread's copy of the variable in the common block becomes uninitialized before execution reaches the parallel region, then each thread except the master thread will initialize the variable to its initial value. The master thread's copy will remain undefined.
22. If a **THREADPRIVATE** common block is specified in a **COPYIN** clause, each new thread duplicates the master thread's definition and association status of the variables in the common block. Therefore, if the master thread's copy of a variable is undefined on entry to a parallel region, each new thread's copy of the variable will also be undefined.

Allocation Status

The allocation status of an allocatable array is one of the following during program execution:

1. Not currently allocated, which means that the array has never been allocated or that the last operation on it was a deallocation.
2. Currently allocated, which means that the array has been allocated by an **ALLOCATE** statement and has not been subsequently deallocated.
3. Undefined, which means that the array does not have the **SAVE** or **STATIC** attribute and was currently allocated when execution of a **RETURN** or **END** statement resulted in no executing scoping units having access to it. This status is only available when you are using the **-qxlf90=noautodealloc** option. (For example, you are using the **xlf90** compilation command.)

In Fortran 95, the allocation status of an allocatable array that is declared in the scope of a module is processor dependent if it does not have the **SAVE** attribute and was currently allocated when execution of a **RETURN** or **END** statement resulted in no executing scoping units referencing the module.

In XL Fortran, the allocation status of such an array remains currently allocated.

If the allocation status of an allocatable array is currently allocated, the array may be referenced and defined. An allocatable array that is not currently allocated must not be referenced or defined. If the allocation status of an allocatable array is undefined, the array must not be referenced, defined, allocated, or deallocated.

Storage Classes for Variables

Note: This section pertains only to storage for variables. Named constants and their subobjects have a storage class of *literal*.

Fundamental Storage Classes

All variables are ultimately represented by one of five storage classes:

Automatic
: for variables in a procedure that will not be retained once the procedure ends. Variables reside in the stack storage area.

Static
: for variables that retain memory throughout the program. Variables reside in the data storage area. Large, uninitialized variables reside in the .bss storage area.

Common
: for common block variables. If a common block variable is initialized, the whole block resides in the data storage area; otherwise, the whole block resides in the .bss storage area.

Controlled Automatic
: for automatic objects. Variables reside in the stack storage area. XL Fortran allocates storage on entry to the procedure and deallocates the storage when the procedure completes.

Controlled
: for allocatable arrays. Variables reside in the heap storage area. You must explicitly allocate and deallocate the storage.

Secondary Storage Classes

None of the following storage classes own their own storage, but are associated with a fundamental storage class at run time.

Pointee
: is dependent on the value of the corresponding integer pointer.

Reference parameter
: is a dummy argument whose actual argument is passed to a procedure using the default passing method or **%REF**.

Value parameter
: is a dummy argument whose actual argument is passed by value to a procedure.

For details on passing methods, see "%VAL and %REF" on page 165.

Storage Class Assignment

Variable names are assigned storage classes in three ways:

1. Explicitly:
   - Dummy arguments have an explicit storage class of reference parameter or value parameter. See "%VAL and %REF" on page 165 for more details.
   - Pointee variables have an explicit storage class of pointee.
   - Variables for which the **STATIC** attribute is explicitly specified have an explicit storage class of static.
   - Variables for which the **AUTOMATIC** attribute is explicitly specified have an explicit storage class of automatic.
   - Variables that appear in a **COMMON** block have an explicit storage class of common.
   - Variables for which the **SAVE** attribute is explicitly specified have an explicit storage class of static, unless they also appear in a **COMMON** statement, in which case their storage class is common.
   - Variables that appear in a **DATA** statement or are initialized in a type declaration statement have an explicit storage class of static, unless they also appear in a **COMMON** statement, in which case their storage class is common.
   - Function result variables that are of type character or derived have the explicit storage class of reference parameter.
   - Function result variables that do not have the **SAVE** or **STATIC** attribute have an explicit storage class of automatic.
   - Automatic objects have an explicit storage class of controlled automatic.
   - Allocatable arrays have an explicit storage class of controlled.

   A variable that does not satisfy any of the above, but that is equivalenced with a variable that has an explicit storage class, inherits that explicit storage class.

   A variable that does not satisfy any of the above, and is not equivalenced with a variable that has an explicit storage class, has an explicit storage class of static if a **SAVE** statement with no list exists in the scoping unit.

2. Implicitly:

   If a variable does not have an explicit storage class, it can be assigned an implicit storage class as follows:
   - Variables whose names begin with a letter, dollar sign or underscore that appears in an **IMPLICIT STATIC** statement have a storage class of static.
   - Variables whose names begin with a letter, dollar sign or underscore that appears in an **IMPLICIT AUTOMATIC** statement have a storage class of automatic.

   In a given scoping unit, if a letter, dollar sign or underscore has not been specified in an **IMPLICIT STATIC** or **IMPLICIT AUTOMATIC** statement, the implicit storage class is the same as that in the host.

   Variables declared in the specification part of a module are associated with the static storage class.

   A variable that does not satisfy any of the above but that is equivalenced with a variable that has an implicit storage class, inherits that implicit storage class.
3. Default:

   All other variables have the default storage class:
   - Static, if you specified the **-qsave** compiler option.
   - Automatic, if you specified the **-qnosave** compiler option. This is the default setting.

     See "-qsave Option" in the *User's Guide* for details on the default settings with regard to the invocation commands.

Chapter 4. Array Concepts

Fortran 90 and Fortran 95 provide a set of features, commonly referred to as array language, that let programmers manipulate arrays. This chapter provides background information on arrays and array language:

- "Arrays"
- "Array Declarators" on page 65
- "Explicit-Shape Arrays" on page 66
- "Assumed-Shape Arrays" on page 68
- "Deferred-Shape Arrays" on page 69
- "Assumed-Size Arrays" on page 71
- "Array Elements" on page 73
- "Array Sections" on page 74
- "Array Constructors" on page 81
- "Expressions Involving Arrays" on page 82

Related Information:

- Many statements in "Chapter 10. Statements" on page 241, have special features and rules for arrays.
- This chapter makes frequent use of the DIMENSION statement. See"DIMENSION" on page 283.
- A number of new intrinsic functions are especially for arrays. These functions are mainly those classified as "Transformational Intrinsic Functions" on page 520.

Arrays

An array is an ordered sequence of scalar data. All the elements of an array have the same type and type parameters.

A *whole array* is denoted by the name of the array:

```
! In this declaration, the array is given a type and dimension
REAL, DIMENSION(3) :: A
! In these expressions, each element is evaluated in each expression
PRINT *, A, A+5, COS(A)
```

A whole array is either a named constant or a variable.

Bounds of a Dimension

Each dimension in an array has an upper and lower bound, which determine the range of values that can be used as subscripts for that dimension. The bound of a dimension can be positive, negative, or zero, in the range -(2**31) to 2**31-1. The range for bounds in 64-bit mode is -(2**63) to 2**63-1.

© Copyright IBM Corp. 1990, 2000

If any lower bound is greater than the corresponding upper bound, the array is a *zero-sized* array, which has no elements but still has the properties of an array. The lower and upper bounds of such a dimension are one and zero, respectively.

When the bounds are specified in array declarators:

- The lower bound is a specification expression. If it is omitted, the default value is 1.
- The upper bound is a specification expression or asterisk (*), and has no default value.

Related Information:

"Specification Expressions" on page 88
"LBOUND(ARRAY, DIM)" on page 575
"UBOUND(ARRAY, DIM)" on page 634

Extent of a Dimension

The *extent* of a dimension is the number of elements in that dimension, computed as the value of the upper bound minus the value of the lower bound, plus one.

```
INTEGER, DIMENSION :: X(5)     ! Extent = 5
REAL :: Y(2:4,3:6)             ! Extent in 1st dimension = 3
                               ! Extent in 2nd dimension = 4
```

The minimum extent is zero, in a dimension where the lower bound is greater than the upper bound.

The theoretical maximum extent is 2**31-1, or 2**63-1 in 64-bit, although hardware addressing considerations make it impractical to declare any combination of data objects whose total size (in bytes) exceeds this value.

Different array declarators that are associated by common, equivalence, or argument association can have different ranks and extents.

Rank, Shape, and Size of an Array

The *rank* of an array is the number of dimensions it has:

```
INTEGER, DIMENSION (10) :: A     ! Rank = 1
REAL, DIMENSION (-5:5,100) :: B  ! Rank = 2
```

According to Fortran 95, an array can have from one to seven dimensions.

With XL Fortran, an array can have from one to twenty dimensions.

A scalar is considered to have rank zero.

The *shape* of an array is derived from its rank and extents. It can be represented as a rank-one array where each element is the extent of the corresponding dimension:

```
INTEGER, DIMENSION (10,10) :: A          ! Shape = (/ 10, 10 /)
REAL, DIMENSION (-5:4,1:10,10:19) :: B   ! Shape = (/ 10, 10, 10 /)
```

The *size* of an array is the number of elements in it, equal to the product of the extents of all dimensions:

```
INTEGER A(5)             ! Size = 5
REAL B(-1:0,1:3,4)       ! Size = 2 * 3 * 4 = 24
```

Related Information

- These examples show only simple arrays where all bounds are constants. For instructions on calculating the values of these properties for more complicated kinds of arrays, see the following sections.
- Related intrinsic functions are "SHAPE(SOURCE)" on page 618, and "SIZE(ARRAY, DIM)" on page 623. The rank of an array `A` is `SIZE(SHAPE(A))`.

Array Declarators

An array declarator declares the shape of an array.

You must declare every named array, and no scoping unit can have more than one array declarator for the same name. An array declarator can appear in the following statements: **COMMON**, integer **POINTER**, **STATIC**, **AUTOMATIC**, **DIMENSION**, **ALLOCATABLE**, **POINTER**, **TARGET** and type declaration.

For example:

```
DIMENSION :: A(1:5)         ! Declarator is "(1:5)"
REAL, DIMENSION(1,1:5) :: B ! Declarator is "(1,1:5)"
INTEGER C(10)               ! Declarator is "(10)"
```

Pointers can be scalars, assumed-shape arrays or explicit-shape arrays.

The form of an array declarator is:

```
►►—(—array_spec—)——————————————————————————————————————————►◄
```

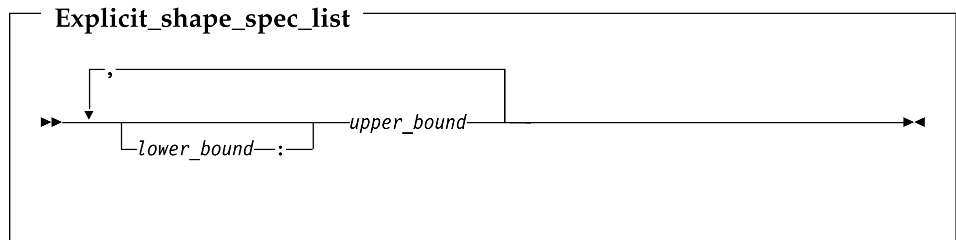
array_spec
: is an array specification. It is a list of dimension declarators, each of which establishes the lower and upper bounds of an array, or specifies that one or both will be set at run time. Each dimension requires one dimension declarator. An array can have from one to twenty dimensions.

An *array_spec* is one of:
explicit_shape_spec_list
assumed_shape_spec_list
deferred_shape_spec_list
assumed_size_spec

Each *array_spec* declares a different kind of array, as explained in the following sections.

Explicit-Shape Arrays

Explicit-shape arrays are arrays where the bounds are explicitly specified for each dimension.



lower_bound, upper_bound
are specification expressions

If any bound is not constant, the array must be declared inside a subprogram and the nonconstant bounds are determined on entry to the subprogram. If a lower bound is omitted, its default value is one.

The rank is the number of specified upper bounds. The shape of an explicit-shape dummy argument can differ from that of the corresponding actual argument.

The size is determined by the specified bounds.

Examples of Explicit-Shape Arrays

```
INTEGER A,B,C(1:10,-5:5)  ! All bounds are constant
A=8; B=3
CALL SUB1(A,B,C)
END
SUBROUTINE SUB1(X,Y,Z)
  INTEGER X,Y,Z(X,Y)      ! Some bounds are not constant
END SUBROUTINE
```

Automatic Arrays

An automatic array is an explicit-shape array that is declared in a subprogram, is not a dummy argument or pointee array, and has at least one

bound that is a nonconstant specification expression. The bounds are evaluated on entry to the subprogram and remain unchanged during execution of the subprogram.

```
INTEGER X
COMMON X
X = 10
CALL SUB1(5)
END

SUBROUTINE SUB1(Y)
  INTEGER X
  COMMON X
  INTEGER Y
  REAL Z (X:20, 1:Y)         ! Automatic array.  Here the bounds are made available
                             ! through dummy arguments and common blocks, although
                             ! Z itself is not a dummy argument.
END SUBROUTINE
```

Related Information

For general information about automatic data objects, see "Automatic Objects" on page 29 and "Storage Classes for Variables" on page 59.

Adjustable Arrays

An *adjustable* array is an explicit-shape array that is declared in a subprogram and has at least one bound that is a nonconstant specification expression. An adjustable array must be a dummy argument.

```
SUBROUTINE SUB1(X, Y)
INTEGER X, Y(X*3)  ! Adjustable array.  Here the bounds depend on a
                   ! dummy argument, and the array name is also passed in.
END SUBROUTINE
```

Pointee Arrays

Pointee arrays are explicit-shape or assumed-size arrays that are declared in integer **POINTER** statements or other specification statements.

The declarator for a pointee array may only contain variables if the array is declared inside a subprogram, and any such variables must be dummy arguments, members of a common block, or use or host associated. The sizes of the dimensions are evaluated upon entry to the subprogram and remain constant during execution of the subprogram.

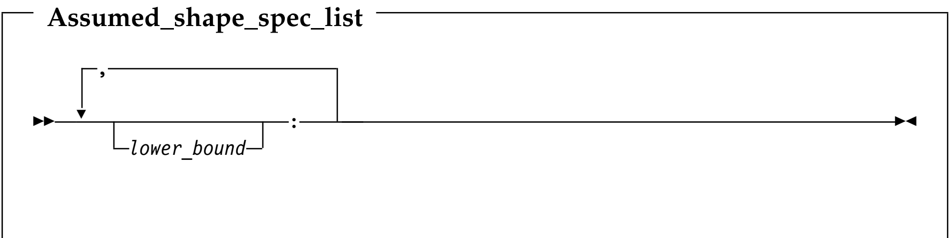
With the **-qddim** compiler option, explained in ″-qddim Option″ in the *User's Guide*, the restrictions on which variables may appear in the array declarator are lifted, declarators in the main program may contain variable names, and any specified nonconstant bounds are re-evaluated each time the array is referenced, so that you can change the properties of the pointee array by simply changing the values of the variables used in the bounds expressions:

```
@PROCESS DDIM
INTEGER PTE, N, ARRAY(10)
POINTER (P, PTE(N))
N = 5
P = LOC(ARRAY(2))  !
PRINT *, PTE       ! Print elements 2 through 6 of ARRAY
N = 7              ! Increase the size
PRINT *, PTE       ! Print elements 2 through 8 of ARRAY
END
```

Related Information:
"POINTER (integer)" on page 366

Assumed-Shape Arrays

Assumed-shape arrays are dummy argument arrays where the extent of each dimension is taken from the associated actual arguments. Because the names of assumed-shape arrays are dummy arguments, they must be declared inside subprograms.



lower_bound is a specification expression

Each lower bound defaults to one, or may be explicitly specified. Each upper bound is set on entry to the subprogram to the specified lower bound (not the lower bound of the actual argument array) plus the extent of the dimension minus one.

The extent of any dimension is the extent of the corresponding dimension of the associated actual argument.

The rank is the number of colons in the *assumed_shape_spec_list*.

The shape is assumed from the associated actual argument array.

The size is determined on entry to the subprogram where it is declared, and equals the size of the associated argument array.

Note: Subprograms that have assumed-shape arrays as dummy arguments must have explicit interfaces.

Examples of Assumed-Shape Arrays

```
INTERFACE
  SUBROUTINE SUB1(B)
    INTEGER B(1:,:,10:)
  END SUBROUTINE
END INTERFACE
INTEGER A(10,11:20,30)
CALL SUB1 (A)
END
SUBROUTINE SUB1(B)
INTEGER B(1:,:,10:)
! Inside the subroutine, B is associated with A.
! It has the same extents as A but different bounds (1:10,1:10,10:39).
END SUBROUTINE
```

Deferred-Shape Arrays

Deferred-shape arrays are allocatable arrays or array pointers, where the bounds can be defined or redefined during execution of the program.

The extent of each dimension (and the related properties of bounds, shape, and size) is undefined until the array is allocated or the pointer is associated with an array that is defined. Before then, no part of the array may be defined, or referenced except as an argument to an appropriate inquiry function. At that point, an array pointer assumes the properties of the target array, and the properties of an allocatable array are specified in an **ALLOCATE** statement.

The rank is the number of colons in the *deferred_shape_spec_list*.

Although a *deferred_shape_spec_list* may sometimes appear identical to an *assumed_shape_spec_list*, deferred-shape arrays and assumed-shape arrays are not the same. A deferred-shape array must have either the **POINTER** attribute or the **ALLOCATABLE** attribute, while an assumed-shape array must be a dummy argument that does not have the **POINTER** attribute. The bounds of a deferred-shape array, and the actual storage associated with it, can be changed at any time by reallocating the array or by associating the pointer

with a different array, while these properties remain the same for an assumed-shape array during the execution of the containing subprogram.

Related Information:

"Allocation Status" on page 58
"Pointer Assignment" on page 117
"ALLOCATABLE" on page 245
"ALLOCATED(ARRAY)" on page 529
"ASSOCIATED(POINTER, TARGET)" on page 533

Allocatable Arrays

A deferred-shape array that has the **ALLOCATABLE** attribute is referred to as an *allocatable array*. Its bounds and shape are determined when storage is allocated for it by an **ALLOCATE** statement.

```
INTEGER, ALLOCATABLE, DIMENSION(:,:,:) :: A
ALLOCATE(A(10,-4:5,20)) ! Bounds of A are now defined (1:10,-4:5,1:20)
DEALLOCATE(A)
ALLOCATE(A(5,5,5))    ! Change the bounds of A
```

Migration Tip:

Minimize storage used:

FORTRAN 77 source

```
INTEGER A(1000),B(1000),C(1000)
C 1000 is the maximum size
WRITE (6,*) "Enter the size of the arrays:"
READ (5,*) N

   ⋮

DO I=1,N
  A(I)=B(I)+C(I)
END DO
END
```

Fortran 90 or Fortran 95 source

```
INTEGER, ALLOCATABLE, DIMENSION(:) :: A,B,C
WRITE (6,*) "Enter the size of the arrays:"
READ (5,*) N
ALLOCATE (A(N),B(N),C(N))

   ⋮

A=B+C
END
```

Related Information:
"Allocation Status" on page 58

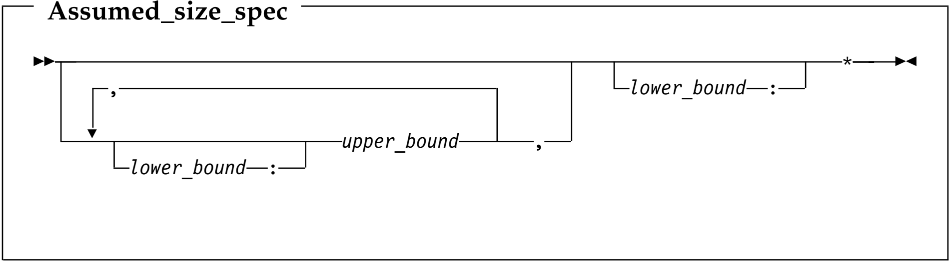
Array Pointers

An array with the **POINTER** attribute is referred to as an array pointer. Its bounds and shape are determined when it is associated with a target through pointer assignment or execution of an **ALLOCATE** statement. It can appear in a type declaration, **POINTER**, or **DIMENSION** statement.

```
REAL, POINTER, DIMENSION(:,:) :: B
REAL, TARGET, DIMENSION(5,10) :: C, D(10:10)
B => C            ! Bounds of B are now defined (1:5,1:10)
B => D            ! B now has different bounds and is associated
                  !   with different storage
ALLOCATE(B(5,5))  ! Change bounds and storage association again
END
```

Assumed-Size Arrays

Assumed-size arrays are dummy argument arrays where the size is inherited from the associated actual array, but the rank and extents may differ. They can only be declared inside subprograms.



lower_bound, upper_bound
are specification expressions

If any bound is not constant, the array must be declared inside a subprogram and the nonconstant bounds are determined on entry to the subprogram. If a lower bound is omitted, its default value is 1.

The last dimension has no upper bound and is designated instead by an asterisk. You must ensure that references to elements do not go past the end of the actual array.

The rank equals one plus the number of *upper_bound* specifications in its declaration, which may be different from the rank of the actual array it is associated with.

The size is assumed from the actual argument that is associated with the assumed-size array:

- If the actual argument is a noncharacter array, the size of the assumed-size array is that of the actual array.
- If the actual argument is an array element from a noncharacter array, and if the size remaining in the array beginning at this element is **S**, then the size of the dummy argument array is **S**. Array elements are processed in array element order.
- If the actual argument is a character array, array element, or array element substring, and assuming that:
  - **A** is the starting offset, in characters, into the character array
  - **T** is the total length, in characters, of the original array
  - **S** is the length, in characters, of an element in the dummy argument array

  then the size of the dummy argument array is:

  MAX(INT (T - A + 1) / S, 0)

  For example:

```
CHARACTER(10) A(10)
CHARACTER(1) B(30)
CALL SUB1(A)           ! Size of dummy argument array is 10
CALL SUB1(A(4))        ! Size of dummy argument array is 6
CALL SUB1(A(6)(5:10))  ! Size of dummy argument array is 3 because there are
                       !   just under 4 elements remaining in A
CALL SUB1(B(12))       ! Size of dummy argument array is 1, because the remainder
                       !   of B can hold just one CHARACTER(10) element
END
SUBROUTINE SUB1(ARRAY)
  CHARACTER(10) ARRAY(*)
  ...
END SUBROUTINE
```

Examples of Assumed-Size Arrays

```
INTEGER X(3,2)
DO I = 1,3
   DO J = 1,2
      X(I,J) = I * J       ! The elements of X are 1, 2, 3, 2, 4, 6
   END DO
END DO
PRINT *,SHAPE(X)           ! The shape is (/ 3, 2 /)
PRINT *,X(1,:)             ! The first row is (/ 1, 2 /)
CALL SUB1(X)
CALL SUB2(X)
END
SUBROUTINE SUB1(Y)
  INTEGER Y(2,*)           ! The dimensions of y are the reverse of x above
  PRINT *, SIZE(Y,1)       ! We can examine the size of the first dimension
                           ! but not the last one.
```

```
  PRINT *, Y(:,1)          ! We can print out vectors from the first
  PRINT *, Y(:,2)          ! dimension, but not the last one.
END SUBROUTINE
SUBROUTINE SUB2(Y)
  INTEGER Y(*)             ! Y has a different rank than X above.
  PRINT *, Y(6)            ! We have to know (or compute) the position of
                           ! the last element.  Nothing prevents us from
                           ! subscripting beyond the end.
END SUBROUTINE
```

Notes:

1. An assumed-size array cannot be used as a whole array in an executable construct unless it is an actual argument in a subprogram reference that does not require the shape:

```
! A is an assumed-size array.

PRINT *, UBOUND(A,1) ! OK - only examines upper bound of first dimension.
PRINT *, LBOUND(A)   ! OK - only examines lower bound of each dimension.
! However, 'B=UBOUND(A)' or 'A=5' would reference the upper bound of
! the last dimension and are not allowed.  SIZE(A) and SHAPE(A) are
! also not allowed.
```

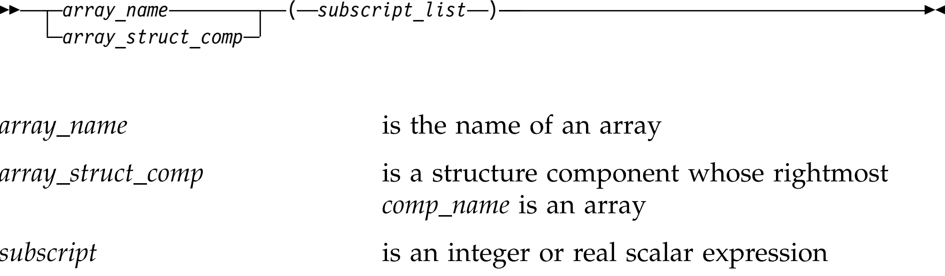
2. If a section of an assumed-size array has a subscript triplet as its last section subscript, the upper bound must be specified. (Array sections and subscript triplets are explained in a subsequent section.)

```
! A is a 2-dimensional assumed-size array
PRINT *, A(:, 6)      ! Triplet with no upper bound is not last dimension.
PRINT *, A(1, 1:10)   ! Triplet in last dimension has upper bound of 10.
PRINT *, A(5, 5:9:2)  ! Triplet in last dimension has upper bound of 9.
```

Array Elements

Array elements are the scalar data that make up an array. Each element inherits the type, type parameters, and **INTENT**, **PARAMETER**, and **TARGET** attributes from its parent array. The **POINTER** attribute is not inherited.

You identify an array element by an *array element designator*, whose form is:



array_name	is the name of an array
array_struct_comp	is a structure component whose rightmost *comp_name* is an array
subscript	is an integer or real scalar expression

Notes

- The number of subscripts must equal the number of dimensions in the array.
- If *array_struct_comp* is present, each part of the structure component except the rightmost must have rank zero (that is, must not be an array name or an array section).
- The value of each subscript expression must not be less than the lower bound or greater than the upper bound for the corresponding dimension.

The *subscript value* depends on the value of each subscript expression and on the dimensions of the array. It determines which element of the array is identified by the array element designator.

Related Information:
"Structure Components" on page 43
"Array Sections and Structure Components" on page 79

Array Element Order

The elements of an array are arranged in storage in a sequence known as the *array element order*, in which the subscripts change most rapidly in the first dimension, and subsequently in the remaining dimensions.

For example, an array declared as `A(2, 3, 2)` has the following elements:

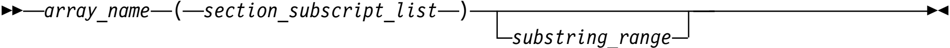
```
Position of Array Element           Array Element Order
-------------------------           -------------------
        A(1,1,1)                            1
        A(2,1,1)                            2
        A(1,2,1)                            3
        A(2,2,1)                            4
        A(1,3,1)                            5
        A(2,3,1)                            6
        A(1,1,2)                            7
        A(2,1,2)                            8
        A(1,2,2)                            9
        A(2,2,2)                           10
        A(1,3,2)                           11
        A(2,3,2)                           12
```

Array Sections

An array section is a selected portion of an array. It is an array subobject that designates a set of elements from an array, or a specified substring or derived-type component from each of those elements. An array section is also an array.

Note: This introductory section describes the simple case, where structure components are not involved. "Array Sections and Structure

Components" on page 79 explains the additional rules for specifying array sections that are also structure components.



section subscript:

```
>>-+-subscript---------+---------------------------------------><
   +-subscript_triplet-+
   '-vector_subscript--'
```

section_subscript
: designates some set of elements along a particular dimension. It can be composed of a combination of the following:

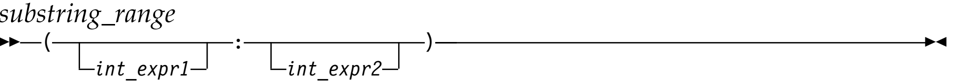
  subscript
  : is a scalar integer or real expression, explained in "Array Elements" on page 73.

  subscript_triplet, vector subscript
  : designate a (possibly empty) sequence of subscripts in a given dimension. For details, see "Subscript Triplets" on page 76 and "Vector Subscripts" on page 78.

  Note: At least one of the dimensions must be a subscript triplet or vector subscript, so that an array section is distinct from an array element:

```
INTEGER, DIMENSION(5,5,5) :: A
A(1,2,3) = 100
A(1,3,3) = 101
PRINT *, A(1,2,3)    ! A single array element, 100.
PRINT *, A(1,2:2,3)  ! A one-element array section, (/ 100 /)
PRINT *, A(1,2:3,3)  ! A two-element array section, (/ 100, 101 /)
```

substring_range



int_expr1 and *int_expr2* are scalar integer expressions called substring expressions, defined in "Character Substrings" on page 37. They specify the leftmost and rightmost character positions, respectively, of a substring of each element in the array section. If an optional *substring_range* is present, the section must be from an array of character objects.

An array section is formed from the array elements specified by the sequences of values from the individual subscripts, subscript triplets, and vector subscripts, arranged in column-major order.

For example, if `SECTION = A( 1:3, (/ 5,6,5 /), 4 )`:
The sequence of numbers for the first dimension is 1, 2, 3.
The sequence of numbers for the second dimension is 5, 6, 5.
The subscript for the third dimension is the constant 4.

The section is made up of the following elements of A, in this order:

```
A(1,5,4)    |                        |        SECTION(1,1)
A(2,5,4)    |----- First column -----|        SECTION(2,1)
A(3,5,4)    |                        |        SECTION(3,1)
A(1,6,4)      |                        |      SECTION(1,2)
A(2,6,4)      |----- Second column ----|      SECTION(2,2)
A(3,6,4)      |                        |      SECTION(3,2)
A(1,5,4)        |                        |    SECTION(1,3)
A(2,5,4)        |----- Third column -----|    SECTION(2,3)
A(3,5,4)        |                        |    SECTION(3,3)
```

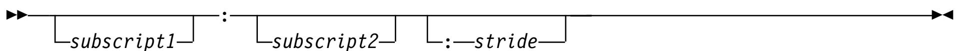
Some examples of array sections include:

```
INTEGER, DIMENSION(10,20) :: A
! These references to array sections require loops or multiple
! statements in FORTRAN 77.
PRINT *, A(1:5,1)                   ! Contiguous sequence of elements
PRINT *, A(1:20:2,10)               ! Noncontiguous sequence of
elements
PRINT *, A(:,5)                     ! An entire column
PRINT *, A( (/1,10,5/), (/7,3,1/) ) ! A 3x3 assortment of elements
```

Related Information:
"Structure Components" on page 43.

Subscript Triplets

A subscript triplet consists of two subscripts and a stride, and defines a sequence of numbers corresponding to array element positions along a single dimension.



subscript1, subscript2
are subscripts that designate the first and last values in the sequence of indices for a dimension.

If the first subscript is omitted, the lower array bound of that dimension is used. If the second subscript is omitted, the upper array bound of that dimension is used. (The second

subscript is mandatory for the last dimension when specifying sections of an assumed-size array.)

stride
: is a scalar integer or real expression that specifies how many subscript positions to count to reach the next selected element.

  If the stride is omitted, it has a value of 1. The stride must have a nonzero value:

  - A positive stride specifies a sequence of integers that begins with the first subscript and proceeds in increments of the stride to the largest integer that is not greater than the second subscript. If the first subscript is greater than the second, the sequence is empty.
  - When the stride is negative, the sequence begins at the first subscript and continues in increments specified by the stride to the smallest integer equal to or greater than the second subscript. If the second subscript is greater than the first, the sequence is empty.

Calculations of values in the sequence use the same steps as shown in "Executing a DO Statement" on page 127.

A subscript in a subscript triplet does not have to be within the declared bounds for that dimension if all the values used in selecting the array elements for the array section are within the declared bounds:

```
INTEGER A(9)
PRINT *, A(1:9:2)  ! Count from 1 to 9 by 2s: 1, 3, 5, 7, 9.
PRINT *, A(1:10:2) ! Count from 1 to 10 by 2s: 1, 3, 5, 7, 9.
                   ! No element past A(9) is specified.
```

Examples of Subscript Triplets

```
REAL, DIMENSION(10) :: A
INTEGER, DIMENSION(10,10) :: B
CHARACTER(10) STRING(1:100)

PRINT *, A(:)                ! Print all elements of array.
PRINT *, A(:5)               ! Print elements 1 through 5.
PRINT *, A(3:)               ! Print elements 3 through 10.

PRINT *, STRING(50:100)      ! Print all characters in
                             ! elements 50 through 100.

! The following statement is equivalent to A(2:10:2) = A(1:9:2)
A(2::2) = A(:9:2)            ! LHS = A(2), A(4), A(6), A(8), A(10)
                             ! RHS = A(1), A(3), A(5), A(7), A(9)
                             ! The statement assigns the odd-numbered
                             ! elements to the even-numbered elements.

! The following statement is equivalent to PRINT *, B(1:4:3,1:7:6)
```

```
PRINT *, B(:4:3,:7:6)         ! Print B(1,1), B(4,1), B(1,7), B(4,7)

PRINT *, A(10:1:-1)           ! Print elements in reverse order.

PRINT *, A(10:1:1)            ! These two are
PRINT *, A(1:10:-1)           ! both zero-sized.
END
```

Vector Subscripts

A vector subscript is an integer or real array expression of rank one, designating a sequence of subscripts that correspond to the values of the elements of the expression.

The sequence does not have to be in order, and may contain duplicate values:

```
INTEGER A(10), B(3), C(3)
PRINT *, A( (/ 10,9,8 /) ) ! Last 3 elements in reverse order
B = A( (/ 1,2,2 /) )       ! B(1) = A(1), B(2) = A(2), B(3) = A(2) also
END
```

An array section with a vector subscript in which two or more elements of the vector subscript have the same value is called a many-one section. Such a section must not:

- Appear on the left side of the equal sign in an assignment statement
- Be initialized through a **DATA** statement
- Be used as an input item in a **READ** statement

Notes:

1. An array section used as an internal file must not have a vector subscript.
2. If you pass an array section with a vector subscript as an actual argument, the associated dummy argument must not be defined or redefined.
3. An array section with a vector subscript must not be the target in a pointer assignment statement.

```
! We can use the whole array VECTOR as a vector subscript for A and B
INTEGER, DIMENSION(3) :: VECTOR= (/ 1,3,2 /), A, B
INTEGER, DIMENSION(4) :: C = (/ 1,2,4,8 /)
A(VECTOR) = B           ! A(1) = B(1), A(3) = B(2), A(2) = B(3)
A = B( (/ 3,2,1 /) )    ! A(1) = B(3), A(2) = B(2), A(3) = B(1)
PRINT *, C(VECTOR(1:2))  ! Prints C(1), C(3)
END
```

Array Sections and Substring Ranges

For an array section with a substring range, each element in the result is the designated character substring of the corresponding element of the array section. The rightmost array name or component name must be of type character.

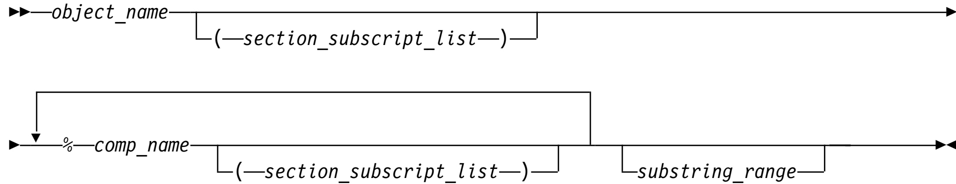
```
PROGRAM SUBSTRING
TYPE DERIVED
  CHARACTER(10) STRING(5)       ! Each structure has 5 strings of 10 chars.
END TYPE DERIVED
```

```
TYPE (DERIVED) VAR, ARRAY(3,3)  ! A variable and an array of derived type.

VAR%STRING(:)(1:3) = 'abc'      ! Assign to chars 1-3 of elements 1-5.
VAR%STRING(3:)(4:6) = '123'     ! Assign to chars 4-6 of elements 3-5.

ARRAY(1:3,2)%STRING(3)(5:10) = 'hello'
                                ! Assign to chars 5-10 of the third element in
                                ! ARRAY(1,2)%STRING, ARRAY(2,2)%STRING, and
END                             ! ARRAY(3,2)%STRING
```

Array Sections and Structure Components

To understand how array sections and structure components overlap, you should be familiar with the syntax for "Structure Components" on page 43.

What we defined at the beginning of this section as an array section is really only a subset of the possible array sections. An array name or array name with a *section_subscript_list* can be a subobject of a structure component:



object_name is the name of an object of derived type

section_subscript_list, substring_range
are the same as defined under "Array Sections" on page 74

comp_name is the name of a derived-type component

Notes:

1. The type of the last component determines the type of the array.
2. Only one part of the structure component may have nonzero rank. Either the rightmost *comp_name* must have a *section_subscript_list* with nonzero rank, or another part must have nonzero rank.
3. Any parts to the right of the part with nonzero rank must not have the **POINTER** attribute.

```
TYPE BUILDING_T
  LOGICAL RESIDENTIAL
END TYPE BUILDING_T

TYPE STREET_T
  TYPE (BUILDING_T) ADDRESS(500)
END TYPE STREET_T

TYPE CITY_T
  TYPE (STREET_T) STREET(100,100)
END TYPE CITY_T
```

```
TYPE (CITY_T) PARIS
TYPE (STREET_T) S
TYPE (BUILDING_T) RESTAURANT
! LHS is not an array section, no subscript triplets or vector subscripts.
PARIS%STREET(10,20) = S
! None of the parts are array sections, but the entire construct
!   is a section because STREET has a nonzero rank and is not
!   the rightmost part.
PARIS%STREET%ADDRESS(100) = BUILDING_T(.TRUE.)

! STREET(50:100,10) is an array section, making the LHS an array section
!   with rank=2, shape=(/51,10/).
! ADDRESS(123) must not be an array section because only one can appear
!   in a reference to a structure component.
PARIS%STREET(50:100,10)%ADDRESS(123)%RESIDENTIAL = .TRUE.
END
```

Rank and Shape of Array Sections

For an array section that is not a subobject of a structure component, the rank is the number of subscript triplets and vector subscripts in the *section_subscript_list*. The number of elements in the shape array is the same as the number of subscript triplets and vector subscripts, and each element in the shape array is the number of integer values in the sequence designated by the corresponding subscript triplet or vector subscript.

For an array section that is a subobject of a structure component, the rank and shape are the same as those of the part of the component that is an array name or array section.

```
DIMENSION :: ARR1(10,20,100)
TYPE STRUCT2_T
  LOGICAL SCALAR_COMPONENT
END TYPE
TYPE STRUCT_T
  TYPE (STRUCT2_T), DIMENSION(10,20,100) :: SECTION
END TYPE

TYPE (STRUCT_T) STRUCT

! One triplet + one vector subscript, rank = 2.
! Triplet designates an extent of 10, vector subscript designates
!   an extent of 3, thus shape = (/ 10,3 /).
ARR1(:, (/ 1,3,4 /), 10) = 0

! One triplet, rank = 1.
! Triplet designates 5 values, thus shape = (/ 5 /).
STRUCT%SECTION(1,10,1:5)%SCALAR_COMPONENT = .TRUE.

! Here SECTION is the part of the component that is an array,
!   so rank = 3 and shape = (/ 10,20,100 /), the same as SECTION.
STRUCT%SECTION%SCALAR_COMPONENT = .TRUE.
```

Array Constructors

An array constructor is a sequence of specified scalar values. It constructs a rank-one array whose element values are those specified in the sequence.

```
►►—(/—ac_value_list—/)——————————————————————————————————————————►◄
```

ac_value
is an expression or implied-**DO** list that provides values for array elements. Each *ac_value* in the array constructor must have the same type and type parameters.

If *ac_value* is:

- A scalar expression, its value specifies an element of the array constructor.
- An array expression, the values of the elements of the expression, in array element order, specify the corresponding sequence of elements of the array constructor.
- An implied-**DO** list, it is expanded to form an *ac_value* sequence under the control of the *ac_do_variable*, as in the **DO** construct.

The data type of the array constructor is the same as the data type of the *ac_value_list* expressions. If every expression in an array constructor is a constant expression, the array constructor is a constant expression.

You can construct arrays of rank greater than one using an intrinsic function. See "RESHAPE(SOURCE, SHAPE, PAD, ORDER)" on page 613 for details.

```
INTEGER, DIMENSION(5) :: A, B, C, D(2,2)
A = (/ 1,2,3,4,5 /)              ! Assign values to all elements in A
A(3:5) = (/ 0,1,0 /)             ! Assign values to some elements
C = MERGE (A, B, (/ T,F,T,T,F /)) ! Construct temporary logical mask

! The array constructor produces a rank-one array, which
!   is turned into a 2x2 array that can be assigned to D.
D = RESHAPE( SOURCE = (/ 1,2,1,2 /), SHAPE = (/ 2,2 /) )

! Here, the constructor linearizes the elements of D in
!   array-element order into a one-dimensional result.
PRINT *, A( (/ D /) )
```

Implied-DO List for an Array Constructor

Implied-**DO** loops in array constructors help to create a regular or cyclic sequence of values, to avoid specifying each element individually.

A zero-sized array of rank one is formed if the sequence of values generated by the loop is empty.

```
►►─(─ac_value_list─,─implied_do_variable─ = ─expr1─,─expr2─┬──────────┬─)──────────►◄
                                                           └─,─expr3─┘
```

implied_do_variable
is a named scalar integer or real variable. In a nonexecutable statement, the type must be integer. Loop processing follows the same rules as for an implied-**DO** in "DATA" on page 277, and uses integer or real arithmetic depending on the type of the implied-**DO** variable.

The variable has the scope of the implied-**DO**, and it must not have the same name as another implied-**DO** variable in a containing array constructor implied-**DO**:

```
M = 0
PRINT *, (/ (M, M=1, 10) /) ! Array constructor implied-DO
PRINT *, M                  ! M still 0 afterwards
PRINT *, (M, M=1, 10)       ! Non-array-constructor implied-DO
PRINT *, M                  ! This one goes to 11
PRINT *, (/ ((M, M=1, 5), N=1, 3) /)
! The result is a 15-element, one-dimensional array.
! The inner loop cannot use N as its variable.
```

expr1, expr2, and *expr3*
are integer or real scalar expressions

```
PRINT *, (/ (I, I = 1, 3) /)
! Sequence is (1, 2, 3)
PRINT *, (/ (I, I = 1, 10, 2) /)
! Sequence is (1, 3, 5, 7, 9)
PRINT *, (/ (I, I+1, I+2, I = 1, 3) /)
! Sequence is (1, 2, 3, 2, 3, 4, 3, 4, 5)

PRINT *, (/ ( (I, I = 1, 3), J = 1, 3 ) /)
! Sequence is (1, 2, 3, 1, 2, 3, 1, 2, 3)

PRINT *, (/ ( (I, I = 1, J), J = 1, 3 ) /)
! Sequence is (1, 1, 2, 1, 2, 3)

PRINT *, (/2,3,(I, I+1, I = 5, 8)/)
! Sequence is (2, 3, 5, 6, 6, 7, 7, 8, 8, 9).
! The values in the implied-DO loop before
!   I=5 are calculated for each iteration of the loop.
```

Expressions Involving Arrays

Arrays can be used in the same kinds of expressions and operations as scalars. Intrinsic operations, assignments, or elemental procedures can be applied to one or more arrays.

For intrinsic operations, in expressions involving two or more array operands, the arrays must have the same shape so that the corresponding elements of each array can be assigned to or be evaluated. In a defined operation arrays can have different shapes. Arrays with the same shape are *conformable*. In a context where a conformable entity is expected, you can also use a scalar value: it is conformable with any array, such that each array element has the value of the scalar.

For example:

```
INTEGER, DIMENSION(5,5) :: A,B,C
REAL, DIMENSION(10) :: X,Y
! Here are some operations on arrays
A = B + C         ! Add corresponding elements of both arrays.
A = -B            ! Assign the negative of each element of B.
A = MAX(A,B,C)    ! A(i,j) = MAX( A(i,j), B(i,j), C(i,j) )
X = SIN(Y)        ! Calculate the sine of each element.
! These operations show how scalars are conformable with arrays
A = A + 5         ! Add 5 to each element.
A = 10            ! Assign 10 to each element.
A = MAX(B, C, 5)  ! A(i,j) = MAX( B(i,j), C(i,j), 5 )

END
```

Related Information:

"Elemental Intrinsic Procedures" on page 519
"Intrinsic Assignment" on page 103
"WHERE" on page 414 shows a way to assign values to some elements in an array but not to others
"FORALL Construct" on page 113

Chapter 5. Expressions and Assignment

This chapter describes the rules for formation, interpretation, and evaluation of expressions and assignment statements:

- "Introduction to Expressions and Assignment"
- "Constant Expressions" on page 87
- "Initialization Expressions" on page 87
- "Specification Expressions" on page 88
- "Operators and Expressions" on page 90
- "Extended Intrinsic and Defined Operations" on page 99
- "How Expressions Are Evaluated" on page 100
- "Intrinsic Assignment" on page 103
- "WHERE Construct" on page 106
- "FORALL Construct" on page 113
- "Pointer Assignment" on page 117

Related Information

- "Defined Operators" on page 150
- "Defined Assignment" on page 151

Introduction to Expressions and Assignment

An expression is a data reference or a computation, and is formed from operands, operators, and parentheses. An expression, when evaluated, produces a value, which has a type, a shape, and possibly type parameters.

An *operand* is either a scalar or an array. An *operator* is either intrinsic or defined. A unary operation has the form:

operator operand

A binary operation has the form:

$operand_1$ *operator* $operand_2$

where the two operands are shape-conforming. If one operand is an array and the other is a scalar, the scalar is treated as an array of the same shape as the array, and every element of the array has the value of the scalar.

Any expression contained in parentheses is treated as a data entity. Parentheses can be used to specify an explicit interpretation of an expression. They can also be used to restrict the alternative forms of the expression, which can help control the magnitude and accuracy of intermediate values during evaluation of the expression. For example, the two expressions

```
(I*J)/K
I*(J/K)
```

© Copyright IBM Corp. 1990, 2000

are mathematically equivalent, but may produce different computational values as a result of evaluation.

Primary

A *primary* is the simplest form of an expression. It can be one of the following:
- A data object
- An array constructor
- A structure constructor
- A complex constructor
- A function reference
- An expression enclosed in parentheses

A primary that is a data object must not be an assumed-size array.

Examples of Primaries

```
12.3             ! Constant
'ABCDEFG'(2:3)   ! Subobject of a constant
VAR              ! Variable name
(/7.0,8.0/)      ! Array constructor
EMP(6,'SMITH')   ! Structure constructor
SIN(X)           ! Function reference
(T-1)            ! Expression in parentheses
```

Type, Parameters, and Shape

The type, type parameters, and shape of a primary are determined as follows:
- A data object or function reference acquires the type, type parameters, and shape of the object or function reference, respectively. The type, parameters, and shape of a generic function reference are determined by the type, parameters, and ranks of its actual arguments.
- A structure constructor is a scalar and its type is that of the constructor name.
- An array constructor has a shape determined by the number of constructor expressions, and its type and parameters are determined by those of the constructor expressions.
- A parenthesized expression acquires the type, parameters, and shape of the expression.

If a pointer appears as a primary in an operation in which it is associated with a nonpointer dummy argument, the target is referenced. The type, parameters, and shape of the primary are those of the target. If the pointer is not associated with a target, it can appear only as an actual argument in a procedure reference whose corresponding dummy argument is a pointer, or as the target in a pointer assignment statement.

Given the operation *[expr1] op expr2*, the shape of the operation is the shape of *expr2* if *op* is unary or if *expr1* is a scalar. Otherwise, its shape is that of *expr1*.

The type and shape of an expression are determined by the operators and by the types and shapes of the expression's primaries. The type of the expression can be intrinsic or derived. An expression of intrinsic type has a kind parameter and, if it is of type character, it also has a length parameter.

Constant Expressions

A *constant expression* is an expression in which each operation is intrinsic and each primary is one of the following:
- A constant or a subobject of a constant.
- An array constructor where each element and the bounds and strides of each implied-**DO** are expressions whose primaries are either constant expressions or implied-**DO** variables.
- A structure constructor where each component is a constant expression.
- An elemental intrinsic function reference where each argument is a constant expression.
- A transformational intrinsic function reference where each argument is a constant expression.
- A reference to the transformational intrinsic function **NULL**.
- A reference to an array inquiry function (except **ALLOCATED**), a numeric inquiry function, the **BIT_SIZE** function, the **LEN** function, or the **KIND** function. Each argument is either a constant expression or it is a variable whose properties inquired about are not assumed, not defined by an expression that is not a constant expression, and not definable by an **ALLOCATE** or pointer assignment statement.
- A constant expression enclosed in parentheses.

Any subscript or substring expression within the expression must be a constant expression.

Examples of Constant Expressions

```
-48.9
name('Pat','Doe')
TRIM('ABC    ')
(MOD(9,4)**3.5)
```

Initialization Expressions

An *initialization expression* is a constant expression. Rules for constant expressions also apply to initialization expressions, except that items that form primaries are constrained by the following rules:

- The exponentiation operation can only have an integer power.

- A primary that is an elemental intrinsic function reference must be of type integer or character, where each argument is an initialization expression of type integer or character.
- Only one of the following transformational functions can be referenced: **REPEAT**, **RESHAPE**, **SELECTED_INT_KIND**, **SELECTED_REAL_KIND**, **TRANSFER**, or **TRIM**. Each argument must be an initialization expression. The following generic intrinsic functions (and related specific functions) are also allowed:
  - ABS (and only the **ABS**, **DABS**, and **QABS** specific functions)
  - AIMAG, IMAG
  - CONJG
  - DIM (and only the **DIM**, **DDIM**, and **QDIM** specific functions)
  - DPROD
  - INT, REAL, DBLE, QEXT, CMPLX, DCMPLX, QCMPLX
  - MAX
  - MIN
  - MOD
  - NINT
  - NULL
  - SIGN
  - INDEX, SCAN, VERIFY (optional 3rd argument allowed)

If an initialization expression includes a reference to an inquiry function for a type parameter or an array bound of an object specified in the same specification part, the type parameter or array bound must be specified in a prior specification of the specification part. The prior specification can be to the left of the inquiry function in the same statement.

Examples of Initialization Expressions

```
3.4**3
KIND(57438)
(/'desk','lamp'/)
'ab'//'cd'//'ef'
```

Specification Expressions

A specification expression is an expression with limitations that you can use to specify items such as character lengths and array bounds.

A *specification expression* is a scalar, integer, restricted expression.

A *restricted expression* is an expression in which each operation is intrinsic and each primary is:

- A constant or a subobject of a constant.
- A variable that is a dummy argument that has neither the **OPTIONAL** nor the **INTENT(OUT)** attribute, or a subobject of such a variable.

- A variable that is in a common block, or a subobject of such a variable.
- A variable accessible by use association or host association, or a subobject of such a variable.
- An array constructor where each element and the bounds and strides of each implied-**DO** are expressions whose primaries are either restricted expressions or implied-**DO** variables.
- A structure constructor where each component is a restricted expression.
- A reference to an array inquiry function (except **ALLOCATED**), the bit inquiry function **BIT_SIZE**, the character inquiry function **LEN**, the kind inquiry function **KIND**, or a numeric inquiry function. Each argument is either a restricted expression, or it is a variable whose properties inquired about are not dependent on the upper bound of the last dimension of an assumed-size array, not defined by an expression that is not a restricted expression, or not definable by an **ALLOCATE** statement or by a pointer assignment statement.
- A reference to any remaining intrinsic functions defined in this book where each argument is a restricted expression.
- A reference to a system inquiry function, where any arguments are restricted expressions.

  Any subscript or substring expression must be a restricted expression.
- A reference to a specification function, where any arguments are restricted expressions.

You can use a *specification function* in a specification expression. A function is a specification function if it is a pure function that is not an intrinsic, internal or statement function. A specification function cannot have a dummy procedure argument, and cannot be recursive.

A variable in a specification expression must have its type and type parameters, if any, specified by a previous declaration in the same scoping unit, or by the implicit typing rules in effect for the scoping unit, or by host or use association. If a variable in a specification expression is typed by the implicit typing rules, its appearance in any subsequent type declaration statement must confirm the implied type and type parameters.

If a specification expression includes a reference to an inquiry function for a type parameter or an array bound of an entity specified in the same specification part, the type parameter or array bound must be specified in a prior specification of the specification part. If a specification expression includes a reference to the value of an element of an array specified in the same specification part, the array bounds must be specified in a prior declaration. The prior specification can be to the left of the inquiry function in the same statement.

Examples of Specification Expressions

```
LBOUND(C,2)+6    ! C is an assumed-shape dummy array
ABS(I)*J         ! I and J are scalar integer variables
276/NN(4)        ! NN is accessible through host association
```

The following example shows how a user-defined pure function, `fact`, can be used in the specification expression of an array-valued function result variable:

```
MODULE MOD
CONTAINS
  INTEGER PURE FUNCTION FACT(N)
  INTEGER, INTENT(IN) :: N
  ...
  END FUNCTION FACT
END MODULE MOD

PROGRAM P
  PRINT *, PERMUTE('ABCD')
  CONTAINS
  FUNCTION PERMUTE(ARG)
     USE MOD
     CHARACTER(*), INTENT(IN) :: ARG
     ...
     CHARACTER(LEN(ARG)) :: PERMUTE(FACT(LEN(ARG)))
     ...
  END FUNCTION PERMUTE
END PROGRAM P
```

Operators and Expressions

This section presents the expression levels in the order of evaluation precedence, from least to most.

General

The general form of an expression (*general_expr*) is:

```
>>-+------------------------------+--expr------------------------><
   '-general_expr--defined_binary_op-'
```

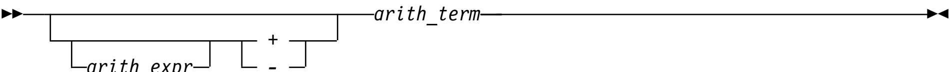
defined_binary_op
: is a defined binary operator. See "Extended Intrinsic and Defined Operations" on page 99.

expr
: is one of the kinds of expressions defined below.

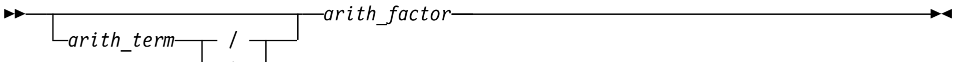
There are four kinds of intrinsic expressions: arithmetic, character, relational, and logical.

Arithmetic

An arithmetic expression (*arith_expr*), when evaluated, produces a numeric value. The form of *arith_expr* is:



The form of *arith_term* is:



The form of *arith_factor* is:

An *arith_primary* is a primary of arithmetic type.

The following table shows the available arithmetic operators and the precedence each takes within an arithmetic expression.

Arithmetic Operator	Representation	Precedence
**	Exponentiation	First
*	Multiplication	Second
/	Division	Second
+	Addition or identity	Third
-	Subtraction or negation	Third

XL Fortran evaluates the terms from left to right when evaluating an arithmetic expression containing two or more addition or subtraction operators. For example, `2+3+4` is evaluated as `(2+3)+4`, although a processor can interpret the expression in another way if it is mathematically equivalent and respects any parentheses.

The factors are evaluated from left to right when evaluating a term containing two or more multiplication or division operators. For example, `2*3*4` is evaluated as `(2*3)*4`.

The primaries are combined from right to left when evaluating a factor containing two or more exponentiation operators. For example, `2**3**4` is evaluated as `2**(3**4)`. (Again, mathematical equivalents are allowed.)

The precedence of the operators determines the order of evaluation when XL Fortran is evaluating an arithmetic expression containing two or more operators having different precedence. For example, in the expression `-A**3`, the exponentiation operator (`**`) has precedence over the negation operator (`-`). Therefore, the operands of the exponentiation operator are combined to form an expression that is used as the operand of the negation operator. Thus, `-A**3` is evaluated as `-(A**3)`.

Note that expressions containing two consecutive arithmetic operators, such as `A**-B` or `A*-B`, are not allowed. You can use expressions such as `A**(-B)` and `A*(-B)`.

If an expression specifies the division of an integer by an integer, the result is rounded to an integer closer to zero. For example, `(-7)/3` has the value `-2`.

For details of exception conditions that can arise during evaluation of floating-point expressions, see ″Detecting and Trapping Floating-Point Exceptions″ in the *User's Guide* .

Examples of Arithmetic Expressions

Arithmetic Expression	Fully Parenthesized Equivalent
-b**2/2.0	-((b**2)/2.0)
i**j**2	i**(j**2)
a/b**2 - c	(a/(b**2)) - c

Data Type of an Arithmetic Expression

Because the identity and negation operators operate on a single operand, the type of the resulting value is the same as the type of the operand.

The following table indicates the resulting type when an arithmetic operator acts on a pair of operands.

Notation: *T(param)*, where *T* is the data type (I: integer, R: real, X: complex) and *param* is the kind type parameter.

Table 3. Result Types for Binary Arithmetic Operators

	second operand									
first operand	**I(1)**	**I(2)**	**I(4)**	**I(8)**	**R(4)**	**R(8)**	**R(16)**	**X(4)**	**X(8)**	**X(16)**
I(1)	I(1)	I(2)	I(4)	I(8)	R(4)	R(8)	R(16)	X(4)	X(8)	X(16)
I(2)	I(2)	I(2)	I(4)	I(8)	R(4)	R(8)	R(16)	X(4)	X(8)	X(16)

Table 3. Result Types for Binary Arithmetic Operators (continued)

	second operand									
first operand	**I(1)**	**I(2)**	**I(4)**	**I(8)**	**R(4)**	**R(8)**	**R(16)**	**X(4)**	**X(8)**	**X(16)**
I(4)	I(4)	I(4)	I(4)	I(8)	R(4)	R(8)	R(16)	X(4)	X(8)	X(16)
I(8)	I(8)	I(8)	I(8)	I(8)	R(4)	R(8)	R(16)	X(4)	X(8)	X(16)
R(4)	R(4)	R(4)	R(4)	R(4)	R(4)	R(8)	R(16)	X(4)	X(8)	X(16)
R(8)	R(8)	R(8)	R(8)	R(8)	R(8)	R(8)	R(16)	X(8)	X(8)	X(16)
R(16)	R(16)	R(16)	R(16)	R(16)	R(16)	R(16)	R(16)	X(16)	X(16)	X(16)
X(4)	X(4)	X(4)	X(4)	X(4)	X(4)	X(8)	X(16)	X(4)	X(8)	X(16)
X(8)	X(8)	X(8)	X(8)	X(8)	X(8)	X(8)	X(16)	X(8)	X(8)	X(16)
X(16)	X(16)	X(16)	X(16)	X(16)	X(16)	X(16)	X(16)	X(16)	X(16)	X(16)

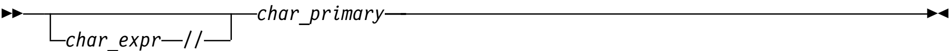
Notes:

1. If you do not specify **-qfloat=rndsngl**, XL Fortran implements **REAL(4)** operations using **REAL(8)** internal precision. If you specify **-qfloat=rndsngl**, XL Fortran implements **REAL(4)** operations using **REAL(4)** internal precision. See ″Detecting and Trapping Floating-Point Exceptions″ in the *User's Guide* for details on modifying this implementation. **REAL(16)** values must only be used in round to nearest mode. The rounding mode can only be changed at the beginning and end of a subprogram. It cannot be changed across a subprogram call; and if it is changed within a subprogram, it must be restored before control is returned to the calling routine.
2. XL Fortran implements integer operations using **INTEGER(4)** arithmetic, or **INTEGER(8)** arithmetic if data items are 8 bytes in length. If the intermediate result is used in a context requiring **INTEGER(1)** or **INTEGER(2)** data type, it is converted as required.

```
INTEGER(2) I2_1, I2_2, I2_RESULT
INTEGER(4) I4
I2_1 = 32767              ! Maximum I(2)
I2_2 = 32767              ! Maximum I(2)
I4 = I2_1 + I2_2
PRINT *, "I4=", I4        ! Prints "I4=65534"

I2_RESULT = I2_1 + I2_2  ! Assignment to I(2) variable
I4 = I2_RESULT            ! and then assigned to an I(4)
PRINT *, "I4=", I4        ! Prints "I4=-2"
END
```

Character

A character expression, when evaluated, produces a result of type character. The form of *char_expr* is:



char_primary is a primary of type character. All character primaries in the expression must have the same kind type parameter, which is also the kind type parameter of the result.

The only character operator is `//`, representing concatenation.

In a character expression containing one or more concatenation operators, the primaries are joined to form one string whose length is equal to the sum of the lengths of the individual primaries. For example, `'AB'//'CD'//'EF'` evaluates to `'ABCDEF'`, a string 6 characters in length.

Parentheses have no effect on the value of a character expression.

A character expression can involve concatenation of an operand whose length was declared with an asterisk in parentheses (indicating inherited length), if the inherited-length character string is used to declare:
- A dummy argument specified in a **FUNCTION**, **SUBROUTINE**, or **ENTRY** statement. The length of the dummy argument assumes the length of the associated actual argument on invocation.
- A named constant. It takes on the length of the constant value.
- The length of an external function result. The calling scoping unit must not declare the function name with an asterisk. On invocation, the length of the function result assumes this defined length.

Example of a Character Expression

```
CHARACTER(7)  FIRSTNAME,LASTNAME
FIRSTNAME='Martha'
LASTNAME='Edwards'
PRINT *, LASTNAME//', '//FIRSTNAME       ! Output:'Edwards, Martha'
END
```

Relational

A relational expression (*rel_expr*), when evaluated, produces a result of type logical, and can appear wherever a logical expression can appear. It can be an arithmetic relational expression or a character relational expression.

Arithmetic Relational Expressions

An arithmetic relational expression compares the values of two arithmetic expressions. Its form is:

```
►►──arith_expr1──relational_operator──arith_expr2──────────────────────────────►◄
```

arith_expr1 and **arith_expr2**
: are each an arithmetic expression. Complex expressions can only be specified if *relational_operator* is **.EQ.**, **.NE.**, <>, ==, or /=.

relational_operator
is any of:

Relational Operator	Representing
.LT. or <	Less than
.LE. or <=	Less than or equal to
.EQ. or ==	Equal to
.NE. or <> or /=	Not equal to
.GT. or >	Greater than
.GE. or >=	Greater than or equal to

An arithmetic relational expression is interpreted as having the logical value `.true.` if the values of the operands satisfy the relation specified by the operator. If the operands do not satisfy the specified relation, the expression has the logical value `.false.`.

If the types or kind type parameters of the expressions differ, their values are converted to the type and kind type parameter of the expression (*arith_expr1* + *arith_expr2*) before evaluation.

Example of an Arithmetic Relational Expression:

```
IF (NODAYS .GT. 365) YEARTYPE = 'leapyear'
```

Character Relational Expressions

A character relational expression compares the values of two character expressions. Its form is:

```
▸▸—char_expr1—relational_operator—char_expr2——————————————————▸◂
```

char_expr1 and *char_expr2*
are each character expressions

relational_operator
is any of the relational operators described in "Arithmetic Relational Expressions" on page 94.

For all relational operators, the collating sequence is used to interpret a character relational expression. The character expression whose value is lower in the collating sequence is less than the other expression. The character expressions are evaluated one character at a time from left to right. You can also use the intrinsic functions (**LGE**, **LLT**, and **LLT**) to compare character strings in the order specified by the ASCII collating sequence. For all relational operators, if the operands are of unequal length, the shorter is

extended on the right with blanks. If both *char_expr1* and *char_expr2* are of zero length, they are evaluated as equal.

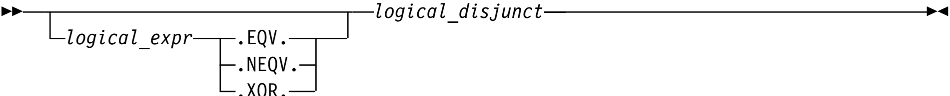
Even if *char_expr1* and *char_expr2* are multibyte characters (MBCS), the ASCII collating sequence is still used.

Example of a Character Relational Expression:

```
IF (CHARIN .GT. '0' .AND. CHARIN .LE. '9') CHAR_TYPE = 'digit'
```

Logical

A logical expression (*logical_expr*), when evaluated, produces a result of type logical. The form of a logical expression is:



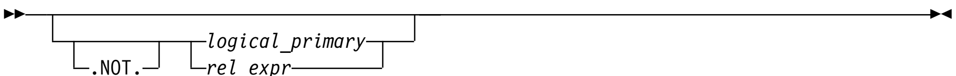
The form of a *logical_disjunct* is:

```
►►─┬──────────────────────────┬─logical_term─────────────────────────►◄
   └─logical_disjunct──.OR.───┘
```

The form of a *logical_term* is:

```
►►─┬────────────────────┬─logical_factor─────────────────────────►◄
   └─logical_term──.AND.─┘
```

The form of a *logical_factor* is:



logical_primary is a primary of type logical.

rel_expr is a relational expression.

The logical operators are:

Logical Operator	Representing	Precedence
.NOT.	Logical negation	First (highest)
.AND.	Logical conjunction	Second
.OR.	Logical inclusive disjunction	Third
.XOR.	Logical exclusive disjunction	Fourth (lowest)
.EQV.	Logical equivalence	Fourth (lowest)
.NEQV.	Logical nonequivalence	Fourth (lowest)

The **.XOR.** operator is treated as an intrinsic operator only when the **-qxlf77=intxor** compiler option is specified. (See ″-qxlf77 Option″ in the *User's Guide* for details.) Otherwise, it is treated as a defined operator. If it is treated as an intrinsic operator, it can also be extended by a generic interface.

The precedence of the operators determines the order of evaluation when a logical expression containing two or more operators having different precedences is evaluated. For example, evaluation of the expression `A.OR.B.AND.C` is the same as evaluation of the expression `A.OR.(B.AND.C)`.

Value of a Logical Expression

Given that `x1` and `x2` represent logical values, use the following tables to determine the values of logical expressions:

x1	.NOT. x1
True	False
False	True

x1	x2	.AND.	.OR.	.XOR.	.EQV.	.NEQV.
False	False	False	False	False	True	False
False	True	False	True	True	False	True
True	False	False	True	True	False	True
True	True	True	True	False	True	False

Sometimes a logical expression does not need to be completely evaluated to determine its value. Consider the following logical expression (assume that `LFCT` is a function of type logical):

```
A .LT. B .OR. LFCT(Z)
```

If `A` is less than `B`, the evaluation of the function reference is not required to determine that this expression is true.

XL Fortran evaluates a logical expression to a **LOGICAL(n)** or **INTEGER(n)** result, where n is the kind type parameter. The value of n depends on the kind parameter of each operand.

By default, for the unary logical operator **.NOT.**, n will be the same as the kind type parameter of the operand. For example, if the operand is **LOGICAL(2)**, the result will also be **LOGICAL(2)**.

The following table shows the resultant type for unary operations:

OPERAND	RESULT of Unary Operation
BYTE	INTEGER(1)
LOGICAL(1)	LOGICAL(1)
LOGICAL(2)	LOGICAL(2)
LOGICAL(4)	LOGICAL(4)
LOGICAL(8)	LOGICAL(8)
Typeless	Default integer

If the operands are of the same length, n will be that length.

For binary logical operations with operands that have different kind type parameters, the kind type parameter of the expression is the same as the larger length of the two operands. For example, if one operand is **LOGICAL(4)** and the other **LOGICAL(2)**, the result will be **LOGICAL(4)**.

The following table shows the resultant type for binary operations:

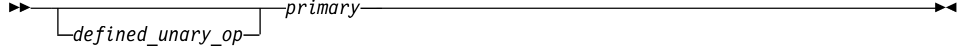
Table 4. Result Types for Binary Logical Expressions

first operand	second operand					
	BYTE	**LOGICAL(1)**	**LOGICAL(2)**	**LOGICAL(4)**	**LOGICAL(8)**	**Typeless**
BYTE	INTEGER(1)	LOGICAL(1)	LOGICAL(2)	LOGICAL(4)	LOGICAL(8)	INTEGER(1)
LOGICAL(1)	LOGICAL(1)	LOGICAL(1)	LOGICAL(2)	LOGICAL(4)	LOGICAL(8)	LOGICAL(1)
LOGICAL(2)	LOGICAL(2)	LOGICAL(2)	LOGICAL(2)	LOGICAL(4)	LOGICAL(8)	LOGICAL(2)
LOGICAL(4)	LOGICAL(4)	LOGICAL(4)	LOGICAL(4)	LOGICAL(4)	LOGICAL(8)	LOGICAL(4)
LOGICAL(8)	LOGICAL(8)	LOGICAL(8)	LOGICAL(8)	LOGICAL(8)	LOGICAL(8)	LOGICAL(8)
Typeless	INTEGER(1)	LOGICAL(1)	LOGICAL(2)	LOGICAL(4)	LOGICAL(8)	Default Integer

If the expression result is to be treated as a default integer but the value cannot be represented within the value range for a default integer, the constant is promoted to a representable kind.

Primary

The form of a primary expression is:



defined_unary_op
is a defined unary operator. See "Extended Intrinsic and Defined Operations".

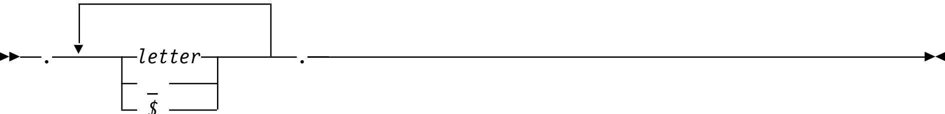
Extended Intrinsic and Defined Operations

A defined operation is either a defined unary operation or a defined binary operation. It is defined by a function and a generic interface block (see "Interface Blocks" on page 144). A defined operation is not an intrinsic operation, although an intrinsic operator can be extended in a defined operation. For example, to add two objects of derived type, you can extend the meaning of the intrinsic binary operator for addition (+). If an extended intrinsic operator has typeless operands, the operation is evaluated intrinsically.

The operand of a unary intrinsic operation that is extended must not have a type that is required by the intrinsic operator. Either or both of the operands of a binary intrinsic operator that is extended must not have the types or ranks that are required by the intrinsic operator.

The defined operator of a defined operation must be defined in a generic interface.

A defined operator is an extended intrinsic operator or has the form:



A defined operator must not contain more than 31 characters and must not be the same as any intrinsic operator or logical literal constant.

See "Generic Interface Blocks" on page 147 for details on defining and extending operators in an interface block.

How Expressions Are Evaluated

Precedence of Operators

An expression can contain more than one kind of operator. When it does, the expression is evaluated from left to right, according to the following precedence among operators:

1. Defined unary
2. Arithmetic
3. Character
4. Relational
5. Logical
6. Defined binary

For example, the logical expression:

```
L .OR. A + B .GE. C
```

where `L` is of type logical, and `A`, `B`, and `C` are of type real, is evaluated the same as the logical expression below:

```
L .OR. ((A + B) .GE. C)
```

An extended intrinsic operator maintains its precedence. That is, the operator does not have the precedence of a defined unary operator or a defined binary operator.

Summary of Interpretation Rules

Primaries that contain operators are combined in the following order:

1. Use of parentheses
2. Precedence of the operators
3. Right-to-left interpretation of exponentiations in a factor
4. Left-to-right interpretation of multiplications and divisions in a term
5. Left-to-right interpretation of additions and subtractions in an arithmetic expression
6. Left-to-right interpretation of concatenations in a character expression
7. Left-to-right interpretation of conjunctions in a logical term
8. Left-to-right interpretation of disjunctions in a logical disjunct
9. Left-to-right interpretation of logical equivalences in a logical expression

Evaluation of Expressions

Arithmetic, character, relational, and logical expressions are evaluated according to the following rules:

- A variable or function must be defined at the time it is used. You must define an integer operand with an integer value, not a statement label value. All referenced characters in a character data object or referenced array elements in an array or array section must be defined at the time the reference is made. All components of a structure must be defined when a structure is referenced. A pointer must be associated with a defined target.

Execution of an array element reference, array section reference, and substring reference requires the evaluation of its subscript, section subscript and substring expressions. Evaluation of any array element subscript, section subscript, substring expression, or the bounds and stride of any array constructor implied-**DO** does not affect, nor is it affected by, the type of the containing expression. See "Expressions Involving Arrays" on page 82. You cannot use any constant integer operation or floating-point operation whose result is not mathematically defined in an executable program. If such expressions are nonconstant and are executed, they are detected at run time. (Examples are dividing by zero and raising a zero-valued primary to a zero-valued or negative-valued power.) As well, you cannot raise a negative-valued primary of type real to a real power.

- The invocation of a function in a statement must not affect, or be affected by, the evaluation of any other entity within the statement in which the function reference appears. When the value of an expression is true, invocation of a function reference in the expression of a logical **IF** statement or a **WHERE** statement can affect entities in the statement that is executed. If a function reference causes definition or undefinition of an actual argument of the function, that argument or any associated entities must not appear elsewhere in the same statement. For example, you cannot use the statements:

```
A(I) = FUNC1(I)
Y = FUNC2(X) + X
```

  if the reference to `FUNC1` defines `I` or the reference to `FUNC2` defines `X`.

  The data type of an expression in which a function reference appears does not affect, nor is it affected by, the evaluation of the actual arguments of the function.

- An argument to a statement function reference must not be altered by evaluating that reference.

Several compiler options affect the data type of the final result:

- When you use the **-qintlog** compiler option, you can mix integer and logical values in expressions and statements. The data type and kind type parameter of the result depends on the operands and the operator involved. In general:
  - For unary logical operators (**.NOT.**) and arithmetic unary operators (**+**,**-**):

Data Type of OPERAND	Data Type of RESULT of Unary Operation
BYTE	INTEGER(1)
INTEGER(n)	INTEGER(n)
LOGICAL(n)	LOGICAL(n)

Data Type of OPERAND	Data Type of RESULT of Unary Operation
Typeless	Default integer

where n represents the kind type parameter. In the case of **INTEGER** and **LOGICAL** data types, the length of the result is the same as the kind type parameter of the operand.

– For binary logical operators (**.AND.**, **.OR.**, **.XOR.**, **.EQV.**, **.NEQV.**) and arithmetic binary operators (**, *, /, +, -), the following table summarizes what data type the result has:

first operand	second operand: BYTE	INTEGER(y)	LOGICAL(y)	Typeless
BYTE	INTEGER(1)	INTEGER(y)	LOGICAL(y)	INTEGER(1)
INTEGER(x)	INTEGER(x)	INTEGER(z)	INTEGER(z)	INTEGER(x)
LOGICAL(x)	LOGICAL(x)	INTEGER(z)	LOGICAL(z)	LOGICAL(x)
Typeless	INTEGER(1)	INTEGER(y)	LOGICAL(y)	Default integer

Note: **z** is the kind type parameter of the result such that **z** is equal to the greater of **x** and **y**. For example, a logical expression with a **LOGICAL(4)** operand and an **INTEGER(2)** operand has a result of **INTEGER(4)**.

For binary logical operators (**.AND.**, **.OR.**, **.XOR.**, **.EQV.**, **.NEQV.**), the result of a logical operation between an integer operand and a logical operand or between two integer operands will be integer. The kind type parameter of the result will be the same as the larger kind parameter of the two operands. If the operands have the same kind parameter, the result has the same kind parameter.

- When you use the **-qlog4** compiler option and the default integer size is **INTEGER(4)**, logical results of logical operations will have type **LOGICAL(4)**, instead of **LOGICAL(n)** as specified in the table above. If you specify the **-qlog4** option and the default integer size is not **INTEGER(4)**, the results will be as specified in the table above.
- When you specify the **-qctyplss** compiler option, XL Fortran treats character constant expressions as Hollerith constants. If one or both operands are character constant expressions, the data type and the length of the result are the same as if the character constant expressions were Hollerith constants. See the ″Typeless″ rows in the previous tables for the data type and length of the result.

See ″XL Fortran Compiler-Option Reference″ in the *User's Guide* for information about compiler options.

Using BYTE Data Objects

Data objects of type **BYTE** can be used wherever a **LOGICAL(1)**, **CHARACTER(1)**, or **INTEGER(1)** data object can be used.

The data types of **BYTE** data objects are determined by the context in which you use them. XL Fortran does not convert them before use. For example, the type of a named constant is determined by use, not by the initial value assigned to it.

- When you use a **BYTE** data object as an operand of an arithmetic, logical, or relational binary operator, the data object assumes:
  - An **INTEGER(1)** data type if the other operand is arithmetic, **BYTE**, or a typeless constant
  - A **LOGICAL(1)** data type if the other operand is logical
  - A **CHARACTER(1)** data type if the other operand is character
- When you use a **BYTE** data object as an operand of the concatenation operator, the data object assumes a **CHARACTER(1)** data type.
- When you use a **BYTE** data object as an actual argument to a procedure with an explicit interface, the data object assumes the type of the corresponding dummy argument:
  - **INTEGER(1)** for an **INTEGER(1)** dummy argument
  - **LOGICAL(1)** for a **LOGICAL(1)** dummy argument
  - **CHARACTER(1)** for a **CHARACTER(1)** dummy argument
- When you use a **BYTE** data object as an actual argument passed by reference to an external subprogram with an implicit interface, the data object assumes a length of 1 byte and no data type.
- When you use a **BYTE** data object as an actual argument passed by value (**%VAL**), the data object assumes an **INTEGER(1)** data type.
- When you use a **BYTE** data object in a context that requires a specific data type, which is arithmetic, logical, or character, the data object assumes an **INTEGER(1)**, **LOGICAL(1)**, or **CHARACTER(1)** data type, respectively.
- A pointer of type **BYTE** cannot be associated with a target of type character, nor can a pointer of type character be associated with a target of type **BYTE**.
- When you use a **BYTE** data object in any other context, the data object assumes an **INTEGER(1)** data type.

Intrinsic Assignment

Assignment statements are executable statements that define or redefine variables based on the result of expression evaluation.

A defined assignment is not intrinsic, and is defined by a subroutine and an interface block. See “Defined Assignment” on page 151.

The general form of an intrinsic assignment is:

```
►►—variable— = —expression—————————————————————————————————►◄
```

The shapes of *variable* and *expression* must conform. *variable* must be an array if *expression* is an array (see "Expressions Involving Arrays" on page 82). If *expression* is a scalar and *variable* is an array, *expression* is treated as an array of the same shape as *variable*, with every array element having the same value as the scalar value of *expression*. *variable* must not be a many-one array section (see "Vector Subscripts" on page 78 for details), and neither *variable* nor *expression* can be an assumed-size array. The types of *variable* and *expression* must conform as follows:

Type of *variable*	**Type of** *expression*
Numeric	Numeric
Logical	Logical
Character	Character
Derived type	Derived type (same as *variable*)

In numeric assignment statements, *variable* and *expression* can specify different numeric types and different kind type parameters. For logical assignment statements, the kind type parameters can differ. For character assignment statements, the length type parameters can differ.

If the length of a character variable is greater than the length of a character expression, the character expression is extended on the right with blanks until the lengths are equal. If the length of the character variable is less than the character expression, the character expression is truncated on the right to match the length of the character variable.

If *variable* is a pointer, it must be associated with a definable target that has type, type parameters and shape that conform with those of *expression*. The value of *expression* is then assigned to the target associated with *variable*.

Both *variable* and *expression* can contain references to any portion of *variable*.

An assignment statement causes the evaluation of *expression* and all expressions within *variable* before assignment, the possible conversion of *expression* to the type and type parameters of *variable*, and the definition of *variable* with the resulting value. No value is assigned to *variable* if it is a zero-length character object or a zero-sized array.

A derived-type assignment statement is an intrinsic assignment statement if there is no accessible defined assignment for objects of this derived type. The

derived type expression must be of the same derived type as the variable. (See "Determining Type for Derived Types" on page 42 for the rules that determine when two structures are of the same derived type.) Assignment is performed as if each component of the expression (or each pointer) is assigned to the corresponding component of the variable. Pointer assignment is executed for pointer components and intrinsic assignment is performed for nonpointer components.

When *variable* is a subobject, the assignment does not affect the definition status or value of other parts of the object.

Arithmetic Conversion

For numeric intrinsic assignment, the value of *expression* may be converted to the type and kind type parameter of *variable*, as specified in the following table:

Type of *variable*	**Value Assigned**
Integer	INT(*expression*,KIND=KIND(*variable*))
Real	REAL(*expression*,KIND=KIND(*variable*))
Complex	CMPLX(*expression*,KIND=KIND(*variable*))

Note: Integer operations for **INTEGER(1)**, **INTEGER(2)**, and **INTEGER(4)** data objects are performed using **INTEGER(4)** arithmetic during evaluation of expressions. If the intermediate result is used in a context requiring an **INTEGER(1)** or **INTEGER(2)** data type, it is converted as required. Integer operations for **INTEGER(8)** data items are performed using **INTEGER(8)** arithmetic. For more information, see item 2 on page 93.

Character Assignment

Only as much of the character expression as is necessary to define the character variable needs to be evaluated. For example:

```
CHARACTER SCOTT*4, DICK*8
SCOTT = DICK
```

This assignment of `DICK` to `SCOTT` requires only that you have previously defined the substring `DICK(1:4)`. You do not have to previously define the rest of `DICK (DICK(5:8))`.

BYTE Assignment

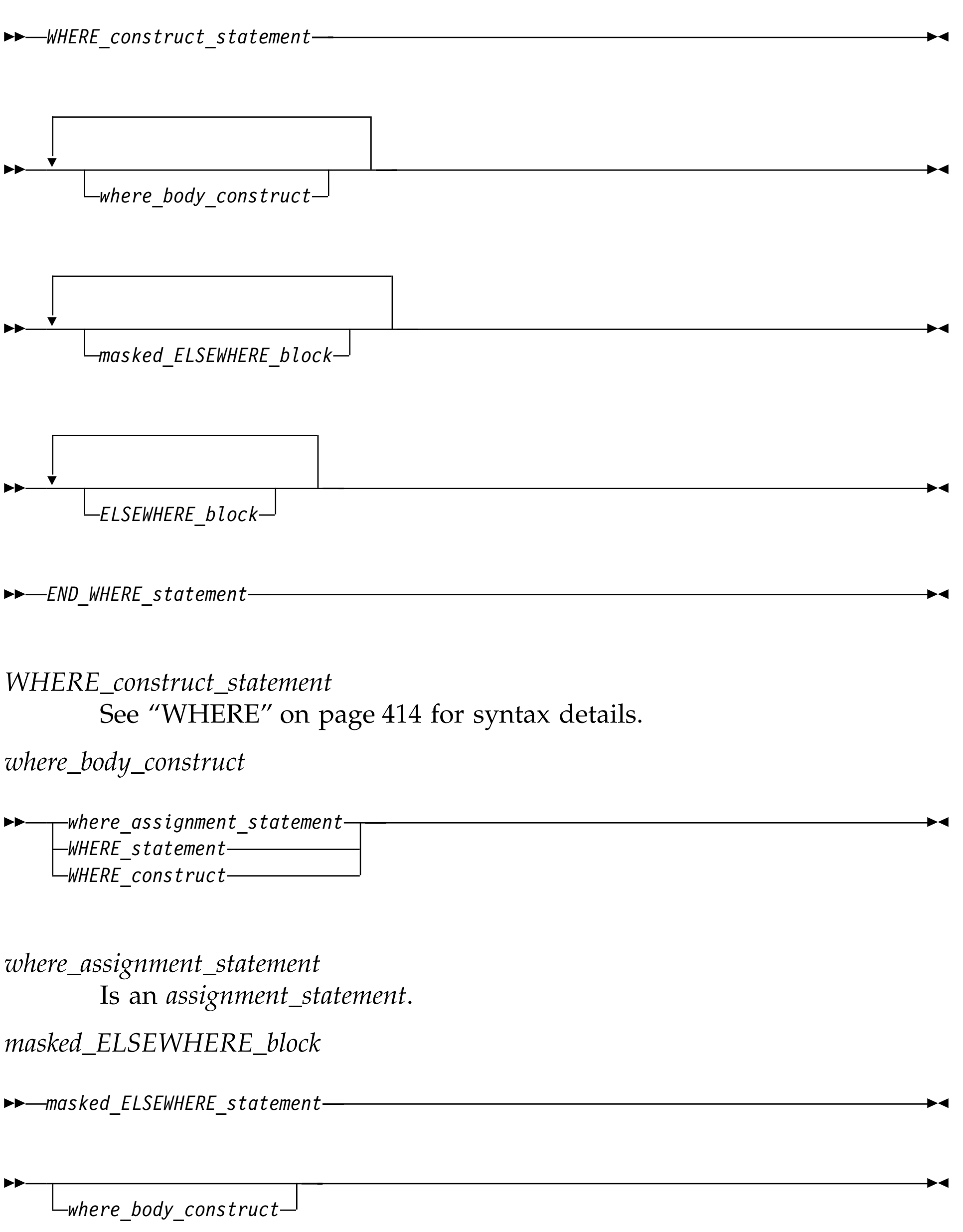
If *expression* is of type arithmetic, arithmetic assignment is used. Similarly, if *expression* is of type character, character assignment is used, and if *expression* is of type logical, logical assignment is used. If the expression on the right is of type **BYTE**, arithmetic assignment is used.

Examples of Intrinsic Assignment:

```
INTEGER I(10)
LOGICAL INSIDE
REAL R,RMIN,RMAX
REAL :: A=2.3,B=4.5,C=6.7
TYPE PERSON
   INTEGER(4) P_AGE
   CHARACTER(20) P_NAME
END TYPE
TYPE (PERSON) EMP1, EMP2
CHARACTER(10) :: CH = 'ABCDEFGHIJ'

I = 5                      ! All elements of I assigned value of 5

RMIN = 28.5 ; RMAX = 29.5
R = (-B + SQRT(B**2 - 4.0*A*C))/(2.0*A)
INSIDE = (R .GE. RMIN) .AND. (R .LE. RMAX)

CH(2:4) = CH(3:5)                 ! CH is now 'ACDEEFGHIJ'

EMP1 = PERSON(45, 'Frank Jones')
EMP2 = EMP1

! EMP2%P_AGE is assigned EMP1%P_AGE using arithmetic assignment
! EMP2%P_NAME is assigned EMP1%P_NAME using character assignment

END
```

WHERE Construct

The **WHERE** construct masks the evaluation of expressions and assignments of values in array assignment statements. It does this according to the value of a logical array expression.



WHERE_construct_statement
: See "WHERE" on page 414 for syntax details.

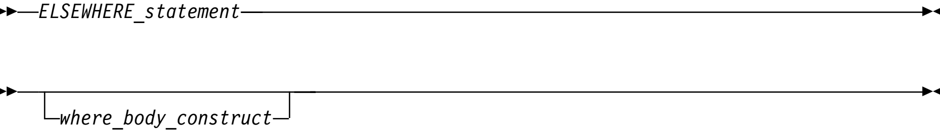
where_body_construct

where_assignment_statement
: Is an *assignment_statement*.

masked_ELSEWHERE_block

masked_ELSEWHERE_statement
: Is an **ELSEWHERE** statement that specifies a *mask_expr*. See "ELSEWHERE" on page 295 for syntax details.

ELSEWHERE_block



ELSEWHERE_statement
: Is an **ELSEWHERE** statement that does not specify a *mask_expr*. See "ELSEWHERE" on page 295 for syntax details.

END_WHERE_statement
: See "END (Construct)" on page 298 for syntax details.

Rules:

- *mask_expr* is a logical array expression.
- A *where_assignment_statement* that is a defined assignment must be an elemental defined assignment.
- In each *where_assignment_statement*, the *mask_expr* and the *variable* being defined must be arrays of the same shape.
- The *mask_expr* on the **WHERE** construct statement and all corresponding masked **ELSEWHERE** statements must have the same shape. The *mask_expr* on a nested **WHERE** statement or nested **WHERE** construct statement must have the same shape as the *mask_expr* on the **WHERE** construct statement of the construct in which it is nested.
- A statement that is part of a *where_body_construct* must not be a branch target statement. Also, **ELSEWHERE**, masked **ELSEWHERE**, and **END WHERE** statements must not be branch target statements.
- If a construct name appears on a **WHERE** construct statement, it must also appear on the corresponding **END WHERE** statement. A construct name is optional on the masked **ELSEWHERE** and **ELSEWHERE** statements in the **WHERE** construct.

Interpreting Masked Array Assignments

To understand how to interpret masked array assignments, you need to understand the concepts of a *control mask* (m_c) and a *pending control mask* (m_p):

- The m_c is an array of type logical whose value determines which elements of an array in a *where_assignment_statement* will be defined. This value is determined by the execution of one of the following: a **WHERE** statement, a **WHERE** construct statement, an **ELSEWHERE** statement, a masked **ELSEWHERE** statement, or an **END WHERE** statement. The value of m_c is cumulative; the compiler determines the value using the mask expressions of surrounding **WHERE** statements and the current mask expression. Subsequent changes to the value of entities in a *mask_expr* have no effect on

the value of m_c. The compiler evaluates the *mask_expr* only once for each **WHERE** statement, **WHERE** construct statement, or masked **ELSEWHERE** statement.

- The m_p is a logical array that provides information to the next masked assignment statement at the same nesting level on the array elements not defined by the current **WHERE** statement, **WHERE** construct statement, or masked **ELSEWHERE** statement.

The following describes how the compiler interprets statements in a **WHERE**, **WHERE** construct, masked **ELSEWHERE**, **ELSEWHERE**, or **END WHERE** statement. It describes the effect on m_c and m_p and any further behavior of the statements, in order of occurrence.

- **WHERE** statement
  - If the **WHERE** statement is nested in a **WHERE** construct, the following occurs:
    1. m_c becomes m_c **.AND.** *mask_expr*.
    2. After the compiler executes the **WHERE** statement, m_c has the value it had prior to the execution of the **WHERE** statement.
  - Otherwise, m_c becomes the *mask_expr*.
- **WHERE** construct
  - If the **WHERE** construct is nested in another **WHERE** construct, the following occurs:
    1. m_p becomes m_c **.AND.** (**.NOT.** *mask_expr*).
    2. m_c becomes m_c **.AND.** *mask_expr*.
  - Otherwise:
    1. The compiler evaluates the *mask_expr*, and assigns m_c the value of that *mask_expr*.
    2. m_p becomes **.NOT.** *mask_expr*.
- Masked **ELSEWHERE** statement

  The following occurs:

  1. m_c becomes m_p.
  2. m_p becomes m_c **.AND.** (**.NOT.** *mask_expr*).
  3. m_c becomes m_c **.AND.** *mask_expr*.
- **ELSEWHERE** statement

  The following occurs:

  1. m_c becomes m_p. No new m_p value is established.
- **END WHERE** statement

  After the compiler executes an **END WHERE** statement, m_c and m_p have the values they had prior to the execution of the corresponding **WHERE** construct statement.

- *where_assignment_statement*

  The compiler assigns the values of the *expr* that correspond to the true values of m_c to the corresponding elements of the *variable*.

If a non-elemental function reference occurs in the *expr* or *variable* of a *where_assignment_statement* or in a *mask_expr*, the compiler evaluates the function without any masked control; that is, it fully evaluates all of the function's argument expressions and then it fully evaluates the function. If the result is an array and the reference is not within the argument list of a non-elemental function, the compiler selects elements corresponding to true values in m_c for use in evaluating the *expr*, *variable*, or *mask_expr*.

If an elemental intrinsic operation or function reference occurs in the *expr* or *variable* of a *where_assignment_statement* or in a *mask_expr*, and is not within the argument list of a non-elemental function reference, the compiler performs the operation or evaluates the function only for the elements corresponding to true values in m_c.

If an array constructor appears in a *where_assignment_statement* or in a *mask_expr*, the compiler evaluates the array constructor without any masked control and then executes the *where_assignment_statement* or evaluates the *mask_expr*.

The execution of a function reference in the *mask_expr* of a **WHERE** statement is allowed to affect entities in the *where_assignment_statement*. Execution of an **END WHERE** has no effect.

The following example shows how control masks are updated. In this example, *mask1*, *mask2*, *mask3*, and *mask4* are conformable logical arrays, m_c is the control mask, and m_p is the pending control mask. The compiler evaluates each mask expression once.

Sample code (with statement numbers shown in the comments):

```
WHERE (mask1)         ! W1
  WHERE (mask2)       ! W2
  ...                 ! W3
  ELSEWHERE (mask3)   ! W4
  ...                 ! W5
  END WHERE           ! W6
ELSEWHERE (mask4)     ! W7
...                   ! W8
ELSEWHERE             ! W9
...                   ! W10
END WHERE             ! W11
```

The compiler sets control and pending control masks as it executes each statement, as shown below:

```
Statement W1
    mc = mask1
    mp = .NOT. mask1
Statement W2
    mp = mask1 .AND. (.NOT. mask2)
    mc = mask1 .AND. mask2
Statement W4
    mc = mask1 .AND. (.NOT. mask2)
    mp = mask1 .AND. (.NOT. mask2)
.AND. (.NOT. mask3)
    mc = mask1 .AND. (.NOT. mask2)
.AND. mask3
Statement W6
    mc = mask1
    mp = .NOT. mask1
Statement W7
    mc = .NOT. mask1
    mp = (.NOT. mask1) .AND. (.NOT.
mask4)
    mc = (.NOT. mask1) .AND. mask4
Statement W9
    mc = (.NOT. mask1) .AND. (.NOT.
mask4)
Statement W11
    mc = 0
    mp = 0
```

The compiler uses the values of the control masks set by statements *W2*, *W4*, *W7*, and *W9* when it executes the respective *where_assignment_statement*s *W3*, *W5*, *W8*, and *W10*.

Migration Tip:

Simplify logical evaluation of arrays

FORTRAN 77 source:

```
INTEGER A(10,10),B(10,10)

   ⋮

DO I=1,10
  DO J=1,10
    IF (A(I,J).LT.B(I,J)) A(I,J)=B(I,J)
  END DO
END DO
END
```

Fortran 90 or Fortran 95 source:

```
INTEGER A(10,10),B(10,10)

   ⋮

WHERE (A.LT.B) A=B
END
```

Examples of the WHERE Construct

```
REAL, DIMENSION(10) :: A,B,C,D
WHERE (A>0.0)
  A = LOG(A)          ! Only the positive elements of A
                      !   are used in the LOG calculation.
  B = A               ! The mask uses the original array A
                      !   instead of the new array A.
  C = A / SUM(LOG(A)) ! A is evaluated by LOG, but
                      !   the resulting array is an
                      !   argument to a non-elemental
                      !   function. All elements in A will
                      !   be used in evaluating SUM.
END WHERE

WHERE (D>0.0)
  C = CSHIFT(A, 1)    ! CSHIFT applies to all elements in array A,
                      ! and the array element values of D determine
                      ! which CSHIFT expression determines the
                      ! corresponding element values of C.
ELSEWHERE
  C = CSHIFT(A, 2)
END WHERE
END
```

The following example shows an array constructor in a **WHERE** construct statement and in a masked **ELSEWHERE** *mask_expr*:

```
CALL SUB((/ 0, -4, 3, 6, 11, -2, 7, 14 /))

CONTAINS
  SUBROUTINE SUB(ARR)
  INTEGER ARR(:)
  INTEGER N

  N = SIZE(ARR)

  ! Data in array ARR at this point:
  !
  ! A = | 0 -4 3 6 11 -2 7 14 |

  WHERE (ARR < 0)
    ARR = 0
  ELSEWHERE (ARR < ARR((/(N-I, I=0, N-1)/)))
    ARR = 2
  END WHERE

  ! Data in array ARR at this point:
  !
  ! A = | 2 0 3 2 11 0 7 14 |

  END SUBROUTINE
END
```

The following example shows a nested **WHERE** construct statement and masked **ELSEWHERE** statement with a *where_construct_name*:

```
INTEGER :: A(10, 10), B(10, 10)
...
OUTERWHERE: WHERE (A < 10)
  INNERWHERE: WHERE (A < 0)
    B = 0
  ELSEWHERE (A < 5) INNERWHERE
    B = 5
  ELSEWHERE INNERWHERE
    B = 10
  END WHERE INNERWHERE
ELSEWHERE OUTERWHERE
  B = A
END WHERE OUTERWHERE
...
```

FORALL Construct

The **FORALL** construct performs assignment to groups of subobjects, especially array elements.

Unlike the **WHERE** construct, **FORALL** performs assignment to array elements, array sections, and substrings. Also, each assignment within a

FORALL construct need not be conformable with the previous one. The **FORALL** construct can contain nested **FORALL** statements, **FORALL** constructs, **WHERE** statements, and **WHERE** constructs.

The **INDEPENDENT** directive specifies that each operation in the **FORALL** statement or construct can be executed in any order without affecting the semantics of the program. For more information on the **INDEPENDENT** directive, see "INDEPENDENT" on page 466.

```
►►──FORALL_construct_statement─────────────────────────────────────────►◄

►►──forall_body────────────────────────────────────────────────────────►◄

►►──END_FORALL_statement───────────────────────────────────────────────►◄
```

FORALL_construct_statement
: See "FORALL (Construct)" on page 314 for syntax details.

END_FORALL_statement
: See "END (Construct)" on page 298 for syntax details.

forall_body
: is one or more of the following statements or constructs:
  - *forall_assignment*
  - **WHERE** statement (see "WHERE" on page 414)
  - **WHERE** construct (see "WHERE Construct" on page 106)
  - **FORALL** statement (see "FORALL" on page 311)
  - **FORALL** construct

forall_assignment
: is either *assignment_statement* or *pointer_assignment_statement*

Any procedures that are referenced in a *forall_body* (including one referenced by a defined operation or defined assignment) must be pure.

If a **FORALL** statement or construct is nested within a **FORALL** construct, the inner **FORALL** statement or construct cannot redefine any *index_name* used in the outer **FORALL** construct.

Although no atomic object can be assigned to, or have its association status changed in the same statement more than once, different assignment statements within the same **FORALL** construct can redefine or reassociate an atomic object. Also, each **WHERE** statement and assignment statement within a **WHERE** construct must follow these restrictions.

If a *FORALL_construct_name* is specified, it must appear in both the **FORALL** statement and the **END FORALL** statement. Neither the **END FORALL** statement nor any statement within the **FORALL** construct can be a branch target statement.

Interpreting the FORALL Construct

1. From the **FORALL** Construct statement, evaluate the *subscript* and *stride* expressions for each *forall_triplet_spec* in any order. All possible pairings of *index_name* values form the set of combinations. For example, given the statement:

   ```
   FORALL (I=1:3,J=4:5)
   ```

   The set of combinations of I and J is:

   ```
   {(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)}
   ```

   The **-1** and **-qnozerosize** compiler options do not affect this step.
2. Evaluate the *scalar_mask_expr* (from the **FORALL** Construct statement) for the set of combinations, in any order, producing a set of active combinations (those that evaluated to .TRUE.). For example, if the mask (I+J.NE.6) is applied to the above set, the set of active combinations is:

   ```
   {(1,4),(2,5),(3,4),(3,5)}
   ```
3. Execute each *forall_body* statement or construct in order of appearance. For the set of active combinations, each statement or construct is executed completely as follows:

   assignment_statement

   Evaluate, in any order, all values in the right-hand side *expression* and all subscripts, strides, and substring bounds in the left-hand side *variable* for all active combinations of *index_name* values.

   Assign, in any order, the computed *expression* values to the corresponding *variable* entities for all active combinations of *index_name* values.

   ```
   INTEGER, DIMENSION(50) :: A,B,C
   INTEGER :: X,I=2,J=49
   FORALL (X=I:J)
     A(X)=B(X)+C(X)
     C(X)=B(X)-A(X) ! All these assignments are performed after the
                    ! assignments in the preceding statement
   END FORALL
   END
   ```

   pointer_assignment_statement

   Determine, in any order, what will be the targets of the pointer assignment, and evaluate all subscripts, strides, and substring bounds in the pointer for all active combinations of *index_name* values. If a target is not a pointer, determination of the target does

not include evaluation of its value. Pointer assignment never *requires* the value of the righthand side to be determined.

Associate, in any order, all targets with the corresponding pointer entities for all active combinations of *index_name* values.

WHERE statement or construct

Evaluate, in any order, the control mask and pending control mask for each **WHERE** statement, **WHERE** construct statement, **ELSEWHERE** statement, or masked **ELSEWHERE** statement each active combination of *index_name* values, producing a refined set of active combinations for that statement, as described in "Interpreting Masked Array Assignments" on page 108. For each active combination, the compiler executes the assignment(s) of the **WHERE** statement, **WHERE** construct statement, or masked **ELSEWHERE** statement for those values of the control mask that are true for that active combination. The compiler executes each statement in a **WHERE** construct in order, as described previously.

```
INTEGER I(100,10), J(100), X
FORALL (X=1:100, J(X)>0)
  WHERE (I(X,:)<0)
    I(X,:)=0       ! Assigns 0 to an element of I along row X only if
                   ! element value is less than 0 and value of element
                   ! in corresponding column of J is greater than 0
  ELSEWHERE
    I(X,:)=1
  END WHERE
END FORALL
END
```

FORALL statement or construct

Evaluate, in any order, the *subscript* and *stride* expressions in the *forall_triplet_spec_list* for the active combinations of the outer **FORALL** statement or construct. The valid combinations are the Cartesian product of combination sets of the inner and outer **FORALL** constructs. The *scalar_mask_expr* determines the active combinations for the inner **FORALL** construct. Statements and constructs for these active combinations are executed.

```
! Same as FORALL (I=1:100,J=1:100,I.NE.J) A(I,J)=A(J,I)

INTEGER A(100,100)
OUTER: FORALL (I=1:100)
  INNER: FORALL (J=1:100,I.NE.J)
    A(I,J)=A(J,I)
  END FORALL INNER
END FORALL OUTER
END
```

Pointer Assignment

The pointer assignment statement causes a pointer to become associated with a target or causes the pointer's association status to become disassociated or undefined.

```
►►—pointer_object— => —target——————————————————————————————————►◄
```

target is a variable or expression. It must have the same type, type parameters and rank as *pointer_object*.

pointer_object must have the **POINTER** attribute.

A target that is an expression must yield a value that has the **POINTER** attribute. A target that is a variable must have the **TARGET** attribute (or be a subobject of such an object) or the **POINTER** attribute. A target must not be an array section with a vector subscript, nor can it be a whole assumed-size array.

The size, bounds, and shape of the target of a disassociated array pointer are undefined. No part of such an array can be defined or referenced, although the array can be the argument of an intrinsic inquiry function that is inquiring about association status, argument presence, or a property of the type or type parameters.

A pointer of type byte can only be associated with a target of type byte, **INTEGER(1)**, or **LOGICAL(1)**.

Any previous association between *pointer_object* and a target is broken. If *target* is not a pointer, *pointer_object* becomes associated with *target*. If *target* is itself an associated pointer, *pointer_object* is associated with the target of *target*. If *target* is a pointer with an association status of disassociated or undefined, *pointer_object* acquires the same status.

Pointer assignment for a pointer structure component can also occur via execution of a derived-type intrinsic assignment statement or a defined assignment statement.

During pointer assignment of an array pointer, the lower bound of each dimension is the result of the **LBOUND** intrinsic function applied to the corresponding dimension of the target. For an array section or array expression that is not a whole array or a structure component, the lower bound is 1. The upper bound of each dimension is the result of the **UBOUND** intrinsic function applied to the corresponding dimension of the target.

Related Information:

See “ALLOCATE” on page 246 for an alternative form of associating a pointer with a target.
See “Pointers as Dummy Arguments” on page 170 for details on using pointers in procedure references.

Examples of Pointer Assignment

```
TYPE T
  INTEGER, POINTER :: COMP_PTR
ENDTYPE T
TYPE(T) T_VAR
INTEGER, POINTER :: P,Q,R
INTEGER, POINTER :: ARR(:)
BYTE, POINTER :: BYTE_PTR
LOGICAL(1), POINTER :: LOG_PTR
INTEGER, TARGET :: MYVAR
INTEGER, TARGET :: DARG(1:5)
P => MYVAR               ! P points to MYVAR
Q => P                   ! Q points to MYVAR
NULLIFY (R)              ! R is disassociated
Q => R                   ! Q is disassociated
T_VAR = T(P)             ! T_VAR%COMP_PTR points to MYVAR
ARR => DARG(1:3)
BYTE_PTR => LOG_PTR
END
```

Integer Pointer Assignment

Integer pointer variables can be:

- Used in integer expressions
- Assigned values as absolute addresses
- Assigned the address of a variable using the **LOC** intrinsic function. (Objects of derived type and structure components must be of sequence-derived type when used with the **LOC** intrinsic function.)

Note that the XL Fortran compiler uses 1-byte arithmetic for integer pointers in assignment statements.

Example of Integer Pointer Assignment

```
INTEGER INT_TEMPLATE
POINTER (P,INT_TEMPLATE)
INTEGER MY_ARRAY(10)
DATA MY_ARRAY/1,2,3,4,5,6,7,8,9,10/
INTEGER, PARAMETER :: WORDSIZE=4

P = LOC(MY_ARRAY)
PRINT *, INT_TEMPLATE          ! Prints '1'
P = P + 4;                     ! Add 4 to reach next element
                               !    because arithmetic is byte-based
PRINT *, INT_TEMPLATE          ! Prints '2'

P = LOC(MY_ARRAY)
```

```
DO I = 1,10
  PRINT *,INT_TEMPLATE
  P = P + WORDSIZE            ! Parameterized arithmetic is suggested
END DO
END
```

Chapter 6. Control

This chapter describes:

- "Statement Blocks"
- "IF Construct"
- "CASE Construct" on page 123
- "DO Construct" on page 126
- "DO WHILE Construct" on page 130
- "Branching" on page 131

You can control your program's execution sequence by constructs containing statement blocks and other executable statements that can alter the normal execution sequence, as defined under "Order of Statements and Execution Sequence" on page 25. The construct descriptions in this chapter do not provide detailed syntax of any construct statements; rather, references are made to the ″Statements″ section.

If a construct is contained in another construct, it must be wholly contained (nested) within that construct. If a statement specifies a construct name, it belongs to that construct; otherwise, it belongs to the innermost construct in which it appears.

Statement Blocks

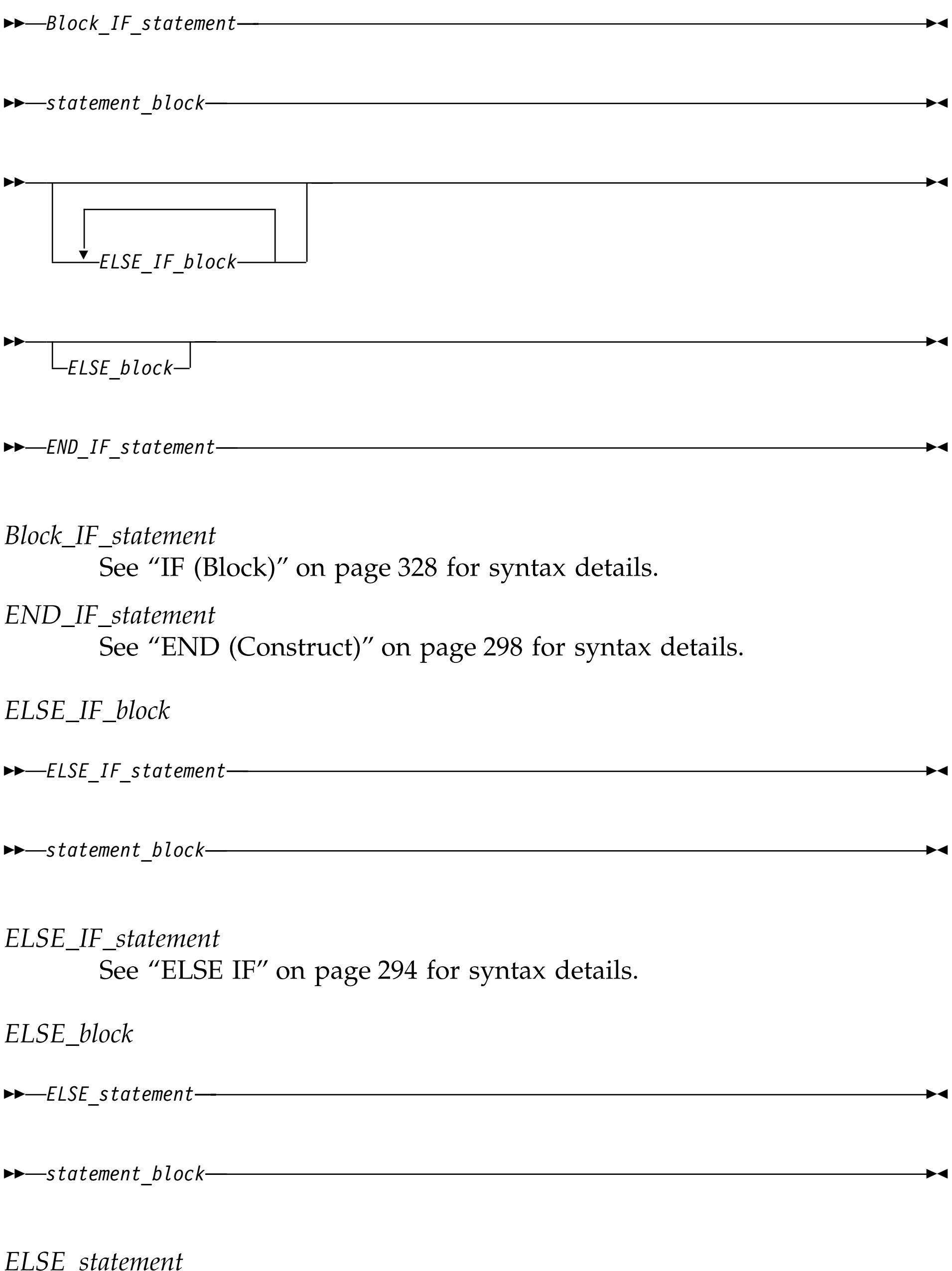
A *statement block* consists of a sequence of zero or more executable statements, executable constructs, **FORMAT** statements, and **DATA** statements that are embedded in another executable construct and are treated as a single unit.

Within an executable program, it is not permitted to transfer control from outside of the statement block to within it. It is permitted to transfer control within the statement block, or from within the statement block to outside the block. For example, in a statement block, you can have a statement with a statement label and a **GO TO** statement using that label.

IF Construct

The **IF** construct selects no more than one of its statement blocks for execution.

© Copyright IBM Corp. 1990, 2000



Block_IF_statement
: See “IF (Block)” on page 328 for syntax details.

END_IF_statement
: See “END (Construct)” on page 298 for syntax details.

ELSE_IF_block

ELSE_IF_statement
: See “ELSE IF” on page 294 for syntax details.

ELSE_block

ELSE_statement
: See “ELSE” on page 294 for syntax details.

The scalar logical expressions in an **IF** construct (that is, the block **IF** and **ELSE IF** statements) are evaluated in the order of their appearance until a true value, an **ELSE** statement, or an **END IF** statement is found:

- If a true value or an **ELSE** statement is found, the statement block immediately following executes, and the **IF** construct is complete. The scalar logical expressions in any remaining **ELSE IF** statements or **ELSE** statements of the **IF** construct are not evaluated.
- If an **END IF** statement is found, no statement blocks execute, and the **IF** construct is complete.

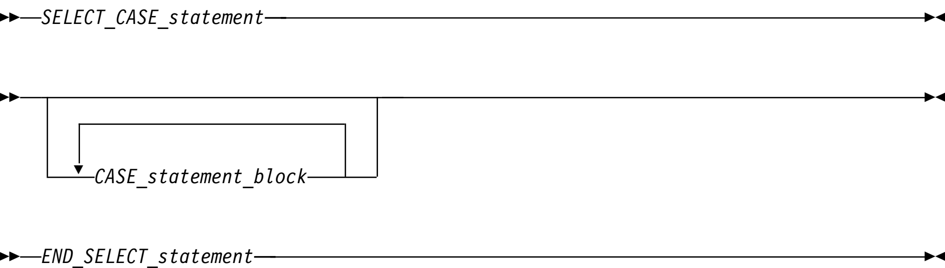
If the **IF** construct name is specified, it must appear on the **IF** statement and **END IF** statement, and optionally on any **ELSE IF** or **ELSE** statements.

Example

```
! Get a record (containing a command) from the terminal

    DO
      WHICHC: IF (CMD .EQ. 'RETRY') THEN       ! named IF construct
           IF (LIMIT .GT. FIVE) THEN           ! nested IF construct
!              Print retry limit exceeded
               CALL STOP
           ELSE
               CALL RETRY
           END IF
      ELSE IF (CMD .EQ. 'STOP') THEN WHICHC    ! ELSE IF blocks
           CALL STOP
      ELSE IF (CMD .EQ. 'ABORT') THEN
           CALL ABORT
      ELSE WHICHC                              ! ELSE block
!          Print unrecognized command
      END IF WHICHC
    END DO
    END
```

CASE Construct

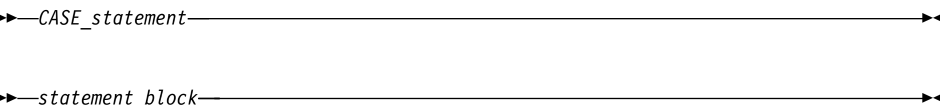
The **CASE** construct has a concise syntax for selecting, at most, one of a number of statement blocks for execution. The case selector of each **CASE** statement is compared to the expression of the **SELECT CASE** statement.



SELECT_CASE_statement
defines the case expression that is to be evaluated. See "SELECT CASE" on page 390 for syntax details.

END_SELECT_statement
terminates the **CASE** construct. See "END (Construct)" on page 298 for syntax details.

CASE_statement_block



CASE_statement
defines the case selector, which is a value, set of values, or default case, for which the subsequent statement block is executed. See "CASE" on page 258 for syntax details.

In the construct, each case value must be of the same type as the case expression.

The **CASE** construct executes as follows:

1. The case expression is evaluated. The resulting value is the case index.
2. The case index is compared to the *case_selector* of each **CASE** statement.
3. If a match occurs, the statement block associated with that **CASE** statement is executed. No statement block is executed if no match occurs. (See "CASE" on page 258.)
4. Execution of the construct is complete and control is transferred to the statement after the **END SELECT** statement.

A **CASE** construct contains zero or more **CASE** statements that can each specify a value range, although the value ranges specified by the **CASE** statements cannot overlap.

A default *case_selector* can be specified by one of the **CASE** statements. A default *CASE_statement_block* can appear anywhere in the **CASE** construct; it can appear at the beginning or end, or among the other blocks.

If a construct name is specified, it must appear on the **SELECT CASE** statement and **END SELECT** statement, and optionally on any **CASE** statements.

You can only branch to the **END SELECT** statement from within the **CASE** construct. A **CASE** statement cannot be a branch target.

Migration Tip:

Use **CASE** in place of block **IF**s.

FORTRAN 77 source

```
IF (I .EQ.3) THEN
     CALL SUBA()
ELSE IF (I.EQ. 5) THEN
     CALL SUBB()
ELSE IF (I .EQ. 6) THEN
     CALL SUBC()
ELSE
     CALL OTHERSUB()
ENDIF
END
```

Fortran 90 or Fortran 95 source

```
SELECTCASE(I)
  CASE(3)
    CALL SUBA()
  CASE(5)
    CALL SUBB()
  CASE(6)
    CALL SUBC()
  CASE DEFAULT
    CALL OTHERSUB()
END SELECT
END
```

Examples

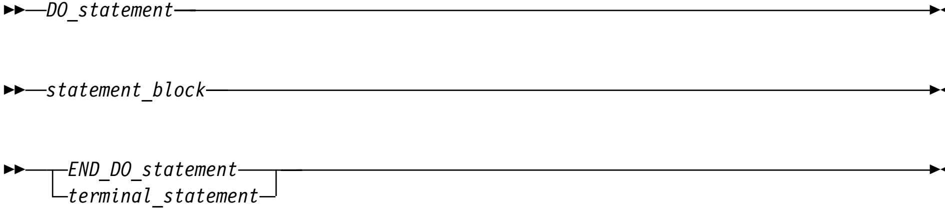
```
ZERO: SELECT CASE(N)

  CASE DEFAULT ZERO
       OTHER: SELECT CASE(N) ! start of CASE construct OTHER
          CASE(:-1)
             SIGNUM = -1     ! this statement executed when n≤-1
          CASE(1:) OTHER
              SIGNUM = 1
       END SELECT OTHER      ! end of CASE construct OTHER
  CASE (0)
    SIGNUM = 0

END SELECT ZERO
END
```

DO Construct

The **DO** construct specifies the repeated execution of a statement block. Such a repeated block is called a *loop*.

The iteration count of a loop can be determined at the beginning of execution of the **DO** construct, unless it is indefinite.

You can curtail a specific iteration with the **CYCLE** statement, and the **EXIT** statement terminates the loop.



DO_statement See "DO" on page 284 for syntax details

END_DO_statement
See "END (Construct)" on page 298 for syntax details

terminal_statement
is a statement that terminates the **DO** construct. See the description below.

If you specify a **DO** construct name on the **DO** statement, you must terminate the construct with an **END DO** statement with the same construct name. Conversely, if you do not specify a **DO** construct name on the **DO** statement, and you terminate the **DO** construct with an **END DO** statement, you must not have a **DO** construct name on the **END DO** statement.

The Terminal Statement

The terminal statement must follow the **DO** statement and must be executable. See "Chapter 10. Statements" on page 241 for a listing of statements that can be used as the terminal statement. If the terminal statement of a **DO** construct is a logical **IF** statement, it can contain any executable statement except those statements to which the restrictions on the logical **IF** statement apply.

If you specify a statement label in the **DO** statement, you must terminate the **DO** construct with a statement that is labeled with that statement label.

You can terminate a labeled **DO** statement with an **END DO** statement that is labeled with that statement label, but you cannot terminate it with an

unlabeled **END DO** statement. If you do not specify a label in the **DO** statement, you must terminate the **DO** construct with an **END DO** statement.

Nested, labeled **DO** and **DO WHILE** constructs can share the same terminal statement if the terminal statement is labeled, and if it is not an **END DO** statement.

Range of a DO Construct

The range of a **DO** construct consists of all the executable statements following the **DO** statement, up to and including the terminal statement. In addition to the rules governing the range of constructs, you can only transfer control to a shared terminal statement from the innermost sharing **DO** construct.

Active and Inactive DO Constructs

A **DO** construct is either active or inactive. Initially inactive, a **DO** construct becomes active only when its **DO** statement is executed. Once active, the **DO** construct becomes inactive only when:

- Its iteration count becomes zero.
- A **RETURN** statement occurs within the range of the **DO** construct.
- Control is transferred to a statement in the same scoping unit but outside the range of the **DO** construct.
- A subroutine invoked from within the **DO** construct returns, through an alternate return specifier, to a statement that is outside the range of the **DO** construct.
- An **EXIT** statement that belongs to the **DO** construct executes.
- An **EXIT** statement or a **CYCLE** statement that is within the range of the **DO** construct, but belongs to an outer **DO** or **DO WHILE** construct, executes.
- A **STOP** statement executes or the program stops for any other reason.

When a **DO** construct becomes inactive, the **DO** variable retains the last value assigned to it.

Executing a DO Statement

An infinite **DO** loops indefinitely.

If the loop is not an infinite **DO**, the **DO** statement includes an initial parameter, a terminal parameter, and an optional increment.

1. The initial parameter, m_1, the terminal parameter, m_2, and the increment, m_3, are established by evaluating the **DO** statement expressions (*a_expr1*, *a_expr2*, and *a_expr3*, respectively). Evaluation includes, if necessary, conversion to the type of the **DO** variable according to the rules for arithmetic conversion. (See "Arithmetic Conversion" on page 105.) If you do not specify *a_expr3*, m_3 has a value of 1. m_3 must not have a value of zero.

2. The **DO** variable becomes defined with the value of the initial parameter (m_1).
3. The iteration count is established, determined by the expression:

   MAX (INT ((m_2 - m_1 + m_3) / m_3), 0)

   Note that the iteration count is 0 whenever:

   $m_1 > m_2$ and $m_3 > 0$, or

   $m_1 < m_2$ and $m_3 < 0$

The iteration count cannot be calculated if the **DO** variable is missing. This is referred to as an infinite **DO** construct.

The iteration count cannot exceed 2**31 - 1 for integer variables of kind 1, 2, or 4, and cannot exceed 2**63 - 1 for integer variables of kind 8. The count becomes undefined if an overflow or underflow situation arises during the calculation.

At the completion of the **DO** statement, loop control processing begins.

Loop Control Processing

Loop control processing determines if further execution of the range of the **DO** construct is required. The iteration count is tested. If the count is not zero, the first statement in the range of the **DO** construct begins execution. If the iteration count is zero, the **DO** construct becomes inactive. If, as a result, all of the **DO** constructs sharing the terminal statement of this **DO** construct are inactive, normal execution continues with the execution of the next executable statement following the terminal statement. However, if some of the **DO** constructs sharing the terminal statement are active, execution continues with incrementation processing of the innermost active **DO** construct.

Execution of the Range

Statements that are part of the statement block are in the range of the **DO** construct. They are executed until the terminal statement is reached. Except by incrementation processing, you cannot redefine the **DO** variable, nor can it become undefined during execution of the range of the **DO** construct.

Terminal Statement Execution

Execution of the terminal statement occurs as a result of the normal execution sequence, or as a result of transfer of control, subject to the restriction that you cannot transfer control into the range of a **DO** construct from outside the range. Unless execution of the terminal statement results in a transfer of control, execution continues with incrementation processing.

Incrementation Processing

1. The **DO** variable, the iteration count, and the increment of the active **DO** construct whose **DO** statement was most recently executed, are selected for processing.
2. The value of the **DO** variable is increased by the value of m_3.
3. The iteration count is decreased by 1.
4. Execution continues with loop control processing of the same **DO** construct whose iteration count was decremented.

Migration Tip:

Use **EXIT**, **CYCLE**, and infinite **DO** statements instead of a **GOTO** statement.

FORTRAN 77 source

```
        I = 0
        J = 0
20      CONTINUE
        I = I + 1
        J = J + 1
        PRINT *, I
        IF (I.GT.4) GOTO 10   ! Exiting loop
        IF (J.GT.3) GOTO 20   ! Iterate loop immediately
        I = I + 2
        GOTO 20
10      CONTINUE
        END
```

Fortran 90 or Fortran 95 source

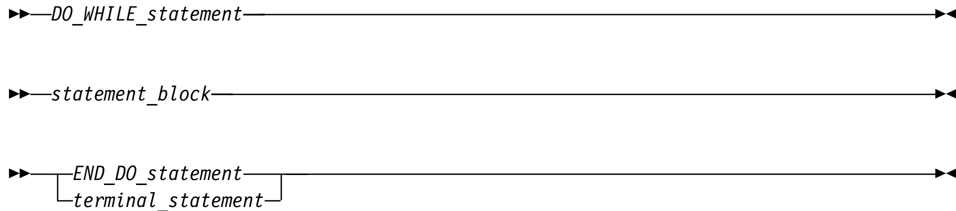
```
        I = 0 ; J = 0
        DO
          I = I + 1
          J = J + 1
          PRINT *, I
          IF (I.GT.4) EXIT
          IF (J.GT.3) CYCLE
          I = I + 2
        END DO
        END
```

Examples:

```
INTEGER :: SUM=0
OUTER: DO
  INNER: DO
    READ (5,*) J
    IF (J.LE.I) THEN
      PRINT *, 'VALUE MUST BE GREATER THAN ', I
      CYCLE INNER
    END IF
    SUM=SUM+J
    IF (SUM.GT.500) EXIT OUTER
    IF (SUM.GT.100) EXIT INNER
  END DO INNER
  SUM=SUM+I
  I=I+10
END DO OUTER
PRINT *, 'SUM =',SUM
END
```

DO WHILE Construct

The **DO WHILE** construct specifies the repeated execution of a statement block for as long as the scalar logical expression specified in the **DO WHILE** statement is true. You can curtail a specific iteration with the **CYCLE** statement, and the **EXIT** statement terminates the loop.



DO_WHILE_statement
: See "DO WHILE" on page 286 for syntax details

END_DO_statement
: See "END (Construct)" on page 298 for syntax details

terminal_stmt
: is a statement that terminates the **DO WHILE** construct. See "The Terminal Statement" on page 126 for details.

The rules discussed earlier concerning **DO** construct names and ranges, active and inactive **DO** constructs, and terminal statements also apply to the **DO WHILE** construct.

Example

```
I=10
TWO_DIGIT: DO WHILE ((I.GE.10).AND.(I.LE.99))
  J=J+I
  READ (5,*) I
END DO TWO_DIGIT
END
```

Branching

You can also alter the normal execution sequence by *branching*. A branch transfers control from one statement to a labeled branch target statement in the same scoping unit. A branch target statement can be any executable statement except a **CASE**, **ELSE**, or **ELSE IF** statement.

The following statements can be used for branching:

- Assigned **GO TO**

  transfers program control to an executable statement, whose statement label is designated in an **ASSIGN** statement. See "GO TO (Assigned)" on page 324 for syntax details.
- Computed **GO TO**

  transfers control to possibly one of several executable statements. See "GO TO (Computed)" on page 325 for syntax details.
- Unconditional **GO TO**

  transfers control to a specified executable statement. See "GO TO (Unconditional)" on page 326 for syntax details.
- Arithmetic **IF**

  transfers control to one of three executable statements, depending on the evaluation of an arithmetic expression. See "IF (Arithmetic)" on page 327 for syntax details.

The following input/output specifiers can also be used for branching:

- the **END=** end-of-file specifier

  transfers control to a specified executable statement if an endfile record is encountered (and no error occurs) in a **READ** statement.
- the **ERR=** error specifier

  transfers control to a specified executable statement in the case of an error. You can specify this specifier in the **BACKSPACE**, **ENDFILE**, **REWIND**, **CLOSE**, **OPEN**, **READ**, **WRITE**, and **INQUIRE** statements.
- the **EOR=** end-or-record specifier

  transfers control to a specified executable statement if an end-of-record condition is encountered (and no error occurs) in a **READ** statement.

Chapter 7. Program Units and Procedures

This chapter describes:

- "Scope"
- "Association" on page 137
- "Program Units, Procedures, and Subprograms" on page 141
- "Interface Blocks" on page 144
- "Generic Interface Blocks" on page 147
- "Main Program" on page 152
- "Modules" on page 153
- "Block Data Program Unit" on page 156
- "Function and Subroutine Subprograms" on page 157
- "Intrinsic Procedures" on page 160
- "Arguments" on page 161
- "Argument Association" on page 163
- "Recursion" on page 175
- "Pure Procedures" on page 176
- "Elemental Procedures" on page 178

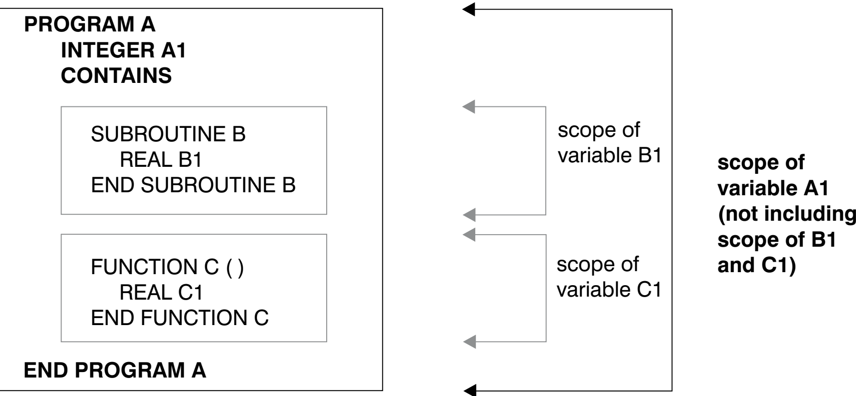
Scope

A program unit consists of a set of nonoverlapping scoping units. A *scoping unit* is that portion of a program unit that has its own scope boundaries. It is one of the following:

- A derived-type definition
- A procedure interface body (not including any derived-type definitions and interface bodies within it)
- A program unit, module subprogram, or internal subprogram (not including derived-type definitions, interface bodies, module subprograms, and internal subprograms).

A *host scoping unit* is the scoping unit that immediately surrounds another scoping unit. For example, in the following diagram, the host scoping unit of the internal function C is the scoping unit of the main program A. Host association is the method by which an internal subprogram, module subprogram, or derived-type definition accesses names from its host.

© Copyright IBM Corp. 1990, 2000



Entities that have scope are:
- A name (see below)
- A label (local entity)
- An external input/output unit number (global entity)
- An operator symbol. Intrinsic operators are global entities, while defined operators are local entities.
- An assignment symbol (global entity)

If the scope is an executable program, the entity is called a *global entity*. If the scope is a scoping unit, the entity is called a *local entity*. If the scope is a statement or part of a statement, the entity is called a *statement entity*. If the scope is a construct, the entity is called a *construct entity*.

The Scope of a Name

Global Entity

Global entities are program units, external procedures, common blocks, and **CRITICAL** *lock_name*s.

If a name identifies a global entity, it cannot be used to identify any other global entity in the same executable program.

See "Conventions for XL Fortran External Names" in the *User's Guide* for details on restrictions on names of global entities.

Local Entity

Entities of the following classes are local entities of the scoping unit in which they are defined:

1. Named variables that are not statement entities, module procedures, named constants, derived-type definitions, construct names, generic identifiers, statement functions, internal subprograms, dummy procedures, intrinsic procedures, or namelist group names.

2. Components of a derived-type definition (each derived-type definition has its own class).

   A component name has the same scope as the type of which it is a component. It may appear only within a component designator of a structure of that type.

   If the derived type is defined in a module and contains the **PRIVATE** statement, the type and its components are accessible in any of the defining module's subprograms by host association. If the accessing scoping unit accesses this type by use association, that scoping unit (and any scoping unit that accesses the entities of that scoping unit by host association) can access the derived-type definition but not its components.
3. Argument keywords (in a separate class for each procedure with an explicit interface).

   A dummy argument name in an internal procedure, module procedure, or procedure interface block has a scope as an argument keyword of the scoping unit of its host. As an argument keyword, it may appear only in a procedure reference for the procedure of which it is a dummy argument. If the procedure or procedure interface block is accessible in another scoping unit by use association or host association, the argument keyword is accessible for procedure references for that procedure in that scoping unit.

In a scoping unit, a name that identifies a local entity of one class may be used to identify a local entity of another class. Such a name must not be used to identify another local entity of the same class, except in the case of generic names. A name that identifies a global entity in a scoping unit cannot be used to identify a local entity of Class 1 in that scoping unit, except for a common block name or the name of an external function.

A common block name in a scoping unit can be the name of any local entity other than a named constant or intrinsic procedure. The name is recognized as the common block entity only when the name is delimited by slashes in a **COMMON**, **VOLATILE**, or **SAVE** statement. If it is not, the name identifies the local entity. An intrinsic procedure name can be the name of a common block in a scoping unit that does not reference the intrinsic procedure. In this case, the intrinsic procedure name is not accessible.

An external function name can also be the function result name. This is the only way that an external function name can also be a local entity.

If a scoping unit contains a local entity of Class 1 with the same name as an intrinsic procedure, the intrinsic procedure is not accessible in that scoping unit.

An interface block generic name can be the same as any of the procedure names in the interface block, or the same as any accessible generic name. It

can be the same as any generic intrinsic procedure. See "Resolution of Procedure References" on page 172 for details.

Statement and Construct Entities

The following items are statement and construct entities:

- Name of a statement function dummy argument.
  SCOPE: Scope of the statement in which it appears.
- Name of a variable that appears as the **DO** variable of an implied-**DO** in a **DATA** statement or array constructor.
  SCOPE: Scope of the implied-**DO** list.
- Name of a variable that appears as an *index_name* in a **FORALL** statement or **FORALL** construct.
  SCOPE: Scope of the **FORALL** statement or construct.

Except for a common block name or scalar variable name, the name of a global entity or local entity of class 1 that is accessible in the scoping unit of a statement or construct must not be the name of a statement or construct entity of that statement or construct. Within the scope of a statement or construct entity, another statement or construct entity must not have the same name.

The name of a variable that appears as a dummy argument in a statement function statement has a scope of the statement in which it appears. It has the type and type parameters that it would have if it were the name of a variable in the scoping unit that includes the statement function.

If the name of a global or local entity accessible in the scoping unit of a statement or construct is the same as the name of a statement or construct entity in that statement or construct, the name is interpreted within the scope of the statement or construct entity as that of the statement or construct entity. Elsewhere in the scoping unit, including parts of the statement or construct outside the scope of the statement or construct entity, the name is interpreted as that of the global or local entity.

If a statement or construct entity has the same name as an accessible name that denotes a variable, constant, or function, the statement or construct entity has the same type and type parameters as the variable, constant or function. Otherwise, the type of the statement or construct entity is determined through the implicit typing rules in effect. If the statement entity is the **DO** variable of an implied-**DO** in a **DATA** statement, the variable cannot have the same name as an accessible named constant. The only attributes held by the **FORALL** statement or construct entities are the type and type parameters that it would have if it were the name of a variable in the scoping unit that includes the **FORALL**. It is type integer.

Except for a common block name or a scalar variable name, a name that identifies a global entity or a local entity of class 1, accessible in the scoping unit of a **FORALL** statement or construct, must not be the same as the *index_name*. Within the scope of a **FORALL** construct, a nested **FORALL** statement or **FORALL** construct must not have the same *index_name*.

If the name of a global or local entity accessible in the scoping unit of a **FORALL** statement or construct is the same as the *index_name*, the name is interpreted within the scope of the **FORALL** statement or construct as that of the *index_name*. Elsewhere in the scoping unit, the name is interpreted as that of the global or local entity.

Association

Association exists if the same data can be identified with different names in the same scoping unit, or with the same name or different names in different scoping units of the same executable program.

Host Association

Host association allows an internal subprogram, module subprogram, or derived-type definition to access named entities that exist in its host. Accessed entities have the same attributes and are known by the same name (if available) as they are in the host. The entities are named objects, derived-type definitions, namelist groups, interface blocks and procedures.

A name that is specified with the **EXTERNAL** attribute is a global name. Any entity in the host scoping unit that has this name as its nongeneric name is inaccessible by that name and by host association.

The following list of entities are local within a scoping unit when declared or initialized in that scoping unit:

- A variable name in a **COMMON** statement or initialized in a **DATA** statement
- An array name in a **DIMENSION** statement or an **ALLOCATABLE** statement
- A name of a derived type
- An object name in a type declaration, **EQUIVALENCE**, **POINTER**, **SAVE**, **TARGET**, **AUTOMATIC**, integer **POINTER**, **STATIC**, or **VOLATILE** statement
- A named constant in a **PARAMETER** statement
- A namelist group name in a **NAMELIST** statement
- A generic interface name or a defined operator
- An intrinsic procedure name in an **INTRINSIC** statement
- A function name in a **FUNCTION** statement, statement function statement, or type declaration statement
- A result name in a **FUNCTION** statement or an **ENTRY** statement

- A subroutine name in a **SUBROUTINE** statement
- An entry name in an **ENTRY** statement
- A dummy argument name in a **FUNCTION**, **SUBROUTINE**, **ENTRY**, or statement function statement
- The name of a named construct

Entities that are local to a subprogram are not accessible in the host scoping unit.

A local entity must not be referenced or defined before the **DATA** statement when:
1. An entity is local to a scoping unit only because it is initialized in a **DATA** statement, and
2. An entity in the host has the same name as this local entity.

If a derived-type name of a host is inaccessible, structures of that type or subobjects of such structures are still accessible.

If a subprogram gains access to a pointer (or integer pointer) by host association, the pointer association that exists at the time the subprogram is invoked remains current within the subprogram. This pointer association can be changed within the subprogram. The pointer association remains current when the procedure finishes executing, except when this causes the pointer to become undefined, in which case the association status of the host-associated pointer becomes undefined.

An interface body does not access named entities through host association, although it can access entities by use association.

The host scoping unit of an internal or module subprogram can contain the same use-associated entities.

Example of Host Association

```
SUBROUTINE MYSUB
TYPE DATES                 ! Define DATES
  INTEGER START
  INTEGER END
END TYPE DATES
CONTAINS
  INTEGER FUNCTION MYFUNC(PNAME)
  TYPE PLANTS
    TYPE (DATES) LIFESPAN    ! Host association of DATES
      CHARACTER(10) SPECIES
      INTEGER PHOTOPER
    END TYPE PLANTS
  END FUNCTION MYFUNC
END SUBROUTINE MYSUB
```

Use Association

Use association occurs when a scoping unit accesses the entities of a module with the **USE** statement. Use-associated entities can be renamed for use in the local scoping unit. The association is in effect for the duration of the executable program. See “USE” on page 408 for details.

```
MODULE M
  CONTAINS
  SUBROUTINE PRINTCHAR(X)
    CHARACTER(20) X
    PRINT *, X
  END SUBROUTINE
END MODULE
PROGRAM MAIN
USE M                        ! Accesses public entities of module M
CHARACTER(20) :: NAME='George'
CALL PRINTCHAR(NAME)         ! Calls PRINTCHAR from module M
END
```

Pointer Association

A target that is associated with a pointer can be referenced by a reference to the pointer. This is called *pointer association*.

A pointer always has an association status:

Associated

- The **ALLOCATE** statement successfully allocates the pointer, which has not been subsequently disassociated or undefined.

  ```
  ALLOCATE (P(3))
  ```
- The pointer is pointer-assigned to a target that is currently associated or has the **TARGET** attribute and, if allocatable, is currently allocated.

  ```
  P => T
  ```

Disassociated

- The pointer is nullified by a **NULLIFY** statement or by the **-qinit=f90ptr** option. See ″-qinit Option″ in the *User's Guide*.

  ```
  NULLIFY (P)
  ```
- The pointer is successfully deallocated.

  ```
  DEALLOCATE (P)
  ```
- The pointer is pointer-assigned to a disassociated pointer.

  ```
  NULLIFY (Q); P => Q
  ```

Undefined

- Initially (unless the **-qinit=f90ptr** option is specified)
- If its target was never allocated.
- If its target was deallocated other than through the pointer.

```
POINTER P(:), Q(:)
ALLOCATE (P(3))
Q => P
DEALLOCATE (Q)   ! Deallocate target of P through Q.
                 ! P is now undefined.
END
```

- If the execution of a **RETURN** or **END** statement causes the pointer's target to become undefined.
- After the execution of a **RETURN** or **END** statement in a procedure where the pointer was declared or accessed, except for objects described in item 4 under "Events Causing Undefinition" on page 55.

Definition Status and Association Status

The definition status of a pointer is that of its target. If a pointer is associated with a definable target, the definition status of the pointer can be defined or undefined according to the rules for a variable.

If the association status of a pointer is disassociated or undefined, the pointer must not be referenced or deallocated. Whatever its association status, a pointer can always be nullified, allocated or pointer-assigned. When it is allocated, its definition status is undefined. When it is pointer-assigned, its association and definition status are determined by its target. So, if a pointer becomes associated with a target that is defined, the pointer becomes defined.

Integer Pointer Association

An integer pointer that is associated with a data object can be used to reference the data object. This is called *integer pointer association*.

Integer pointer association can only occur in the following situations:

- An integer pointer is assigned the address of a variable:

```
POINTER (P,A)
P=LOC(B)              ! A and B become associated
```

- Multiple pointees are declared with the same integer pointer:

```
POINTER (P,A), (P,B)   ! A and B are associated
```

- Multiple integer pointers are assigned the address of the same variable or the address of other variables that are storage associated:

```
POINTER (P,A), (Q,B)
P=LOC(C)
Q=LOC(C)              ! A, B, and C become associated
```

- An integer pointer variable that appears as a dummy argument is assigned the address of another dummy argument or member of a common block:

```
POINTER (P,A)
    .
    .
CALL SUB (P,B)
    .
```

```
      .
SUBROUTINE SUB (P,X)
POINTER (P,Y)
P=LOC(X)                ! Main program variables A
                        !   and B become associated.
```

Program Units, Procedures, and Subprograms

A program unit is a sequence of one or more lines, organized as statements, comments, and **INCLUDE** directives. Specifically, a program unit can be:

- The main program
- A module
- A block data program unit
- An external function subprogram
- An external subroutine subprogram

An executable program is a collection of program units consisting of one main program and any number of external subprograms, modules, and block data program units.

A subprogram can be invoked by a main program or by another subprogram to perform a particular activity. When a procedure is invoked, the referenced subprogram is executed.

An external or module subprogram can contain multiple **ENTRY** statements. The subprogram defines a procedure for the **SUBROUTINE** or **FUNCTION** statement, as well as one procedure for each **ENTRY** statement.

An external procedure is defined either by an external subprogram or by a program unit in a programming language other than Fortran.

Names of main programs, external procedures, block data program units, and modules are global entities. Names of internal and module procedures are local entities.

Internal Procedures

External subprograms, module subprograms, and main programs can have internal subprograms, whether the internal subprograms are functions or subroutines, as long as the internal subprograms follow the **CONTAINS** statement.

An internal procedure is defined by an internal subprogram. Internal subprograms cannot appear in other internal subprograms. A module procedure is defined by a module subprogram or an entry in a module subprogram.

Internal procedures and module procedures are the same as external procedures except that:
- The name of the internal procedure or module procedure is not a global entity
- An internal subprogram must not contain an **ENTRY** statement
- The internal procedure name must not be an argument associated with a dummy procedure
- The internal subprogram or module subprogram has access to host entities by host association

Migration Tip:

Turn your external procedures into internal subprograms or put them into modules. The explicit interface provides type checking.

FORTRAN 77 source

```
PROGRAM MAIN
  INTEGER A
  A=58
  CALL SUB(A)
C A MUST BE PASSED
END
SUBROUTINE SUB(A)
  INTEGER A,B,C   ! A must be redeclared
  C=A+B
END SUBROUTINE
```

Fortran 90 or Fortran 95 source

```
PROGRAM MAIN
  INTEGER :: A=58
  CALL SUB
  CONTAINS
  SUBROUTINE SUB
    INTEGER B,C
    C=A+B         ! A is accessible by host association
  END SUBROUTINE
END
```

Interface Concepts

The interface of a procedure determines the form of the procedure reference. The interface consists of:
- The characteristics of the procedure
- The name of the procedure
- The name and characteristics of each dummy argument
- The generic identifiers of the procedure, if any

The characteristics of a procedure consist of:
- Distinguishing the procedure as a subroutine or a function

- Distinguishing each dummy argument either as a data object, dummy procedure, or alternate return specifier

  The characteristics of a dummy data object are its type, type parameters (if any), shape, intent, whether it is optional, whether it is a pointer, and whether it is a target. Any dependence on other objects for type parameter or array bound determination is a characteristic. If a shape, size, or character length is assumed, it is a characteristic.

  The characteristics of a dummy procedure are the explicitness of its interface, its procedure characteristics (if the interface is explicit), and whether it is optional.
- If the procedure is a function, specifying the characteristics of the result value: its type, type parameters (if any), rank, and whether it is a pointer. For nonpointer array results, its shape is a characteristic. Any dependence on other objects for type parameters or array bound determination is a characteristic. If the length of a character object is assumed, this is a characteristic.

If a procedure is accessible in a scoping unit, it has an interface that is either explicit or implicit in that scoping unit. The rules are:

Entity	**Interface**
Dummy procedure	Explicit in a scoping unit if an interface block exists or is accessible Implicit in all other cases
External subprogram	Explicit in a scoping unit other than its own if an interface block exists or is accessible Implicit in all other cases
Recursive procedure with a result clause	Explicit in the subprogram's own scoping unit
Module procedure	Always explicit
Internal procedure	Always explicit
Generic procedure	Always explicit
Intrinsic procedure	Always explicit
Statement function	Always implicit

Internal subprograms cannot appear in an interface block.

A procedure must not have more than one accessible interface in a scoping unit.

The interface of a statement function cannot be specified in an interface block.

Explicit Interface

A procedure must have an explicit interface if:

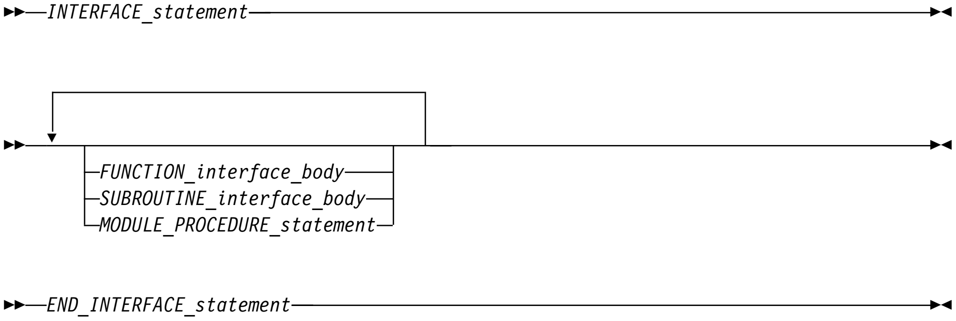
1. A reference to the procedure appears
   - with an argument keyword
   - as a defined assignment (for subroutines only)
   - in an expression as a defined operator (for functions only)
   - as a reference by its generic name
   - in a context that requires it to be pure.
2. The procedure has
   - an optional dummy argument
   - an array-valued result (for functions only)
   - a dummy argument that is a pointer, target, or an assumed-shape array
   - a result whose length type parameter is neither assumed nor constant (for character functions only)
   - a pointer result (for functions only)
3. The procedure is elemental.

Implicit Interface

A procedure has an implicit interface if its interface is not fully known; that is, it has no explicit interface.

Interface Blocks

The *interface block* provides a means of specifying an explicit interface for external procedures and dummy procedures. You can also use an interface block to define generic identifiers. An *interface body* in an interface block specifies the explicit specific interface for an existing external procedure or dummy procedure.

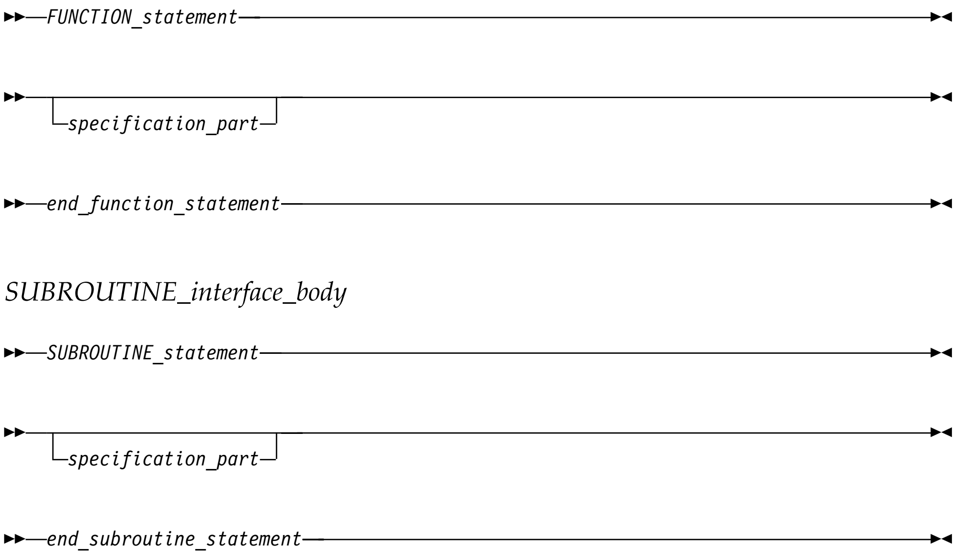


INTERFACE_statement
: See "INTERFACE" on page 344 for syntax details

END_INTERFACE_statement
: See "END INTERFACE" on page 300 for syntax details

MODULE_PROCEDURE_statement
: See “MODULE PROCEDURE” on page 351 for syntax details

FUNCTION_interface_body



FUNCTION_statement, SUBROUTINE_statement
: For syntax details, see “FUNCTION” on page 320 and “SUBROUTINE” on page 396.

specification_part
: is a sequence of statements from the statement groups numbered **2** and **4** in “Order of Statements and Execution Sequence” on page 25.

end_function_statement, end_subroutine_statement
: For syntax details of both statements, see “END” on page 296.

In an interface body, you specify all the characteristics of the procedure. See “Interface Concepts” on page 142. The characteristics must be consistent with those specified in the subprogram definition, except that:

1. dummy argument names may be different.
2. you do not have to indicate that a procedure is pure, even if the subprogram that defines it is pure.
3. you can associate a pure actual argument with a dummy procedure that is not pure.
4. when you associate an intrinsic elemental procedure with a dummy procedure, the dummy procedure does not have to be elemental

The *specification_part* of an interface body can contain statements that specify attributes or define values for data objects that do not determine characteristics of the procedure. Such specification statements have no effect on the interface. Interface blocks do not specify the characteristics of module procedures, whose characteristics are defined in the module subprogram definitions.

An interface body cannot contain **ENTRY** statements, **DATA** statements, **FORMAT** statements, statement function statements, or executable statements. You can specify an entry interface by using the entry name as the procedure name in an interface body.

An interface body does not access named entities by host association. It is treated as if it had a host with the default implicit rules. See "How Type Is Determined" on page 52 for a discussion of the implicit rules.

An interface block can be generic or nongeneric. A generic interface block must specify a generic specification in the **INTERFACE** statement, while a nongeneric interface block must not specify such a generic specification. See "INTERFACE" on page 344 for details.

The interface bodies within a nongeneric interface block can contain interfaces for both subroutines and functions.

A generic name specifies a single name to reference all of the procedures in the interface block. At most, one specific procedure is invoked each time there is a procedure reference with a generic name.

The **MODULE PROCEDURE** statement is allowed only if the interface block has a generic specification and is contained in a scoping unit where each procedure name is accessible as a module procedure.

A procedure name used in a **MODULE PROCEDURE** statement must not have been previously specified in any module procedure statement with the same generic identifier in the same *specification part*.

For an interface to a non-Fortran subprogram, the dummy argument list in the **FUNCTION** or **SUBROUTINE** statement can explicitly specify the passing method. See "Dummy Arguments" on page 162 for details.

Example of an Interface

```
MODULE M
 CONTAINS
 SUBROUTINE S1(IARG)
   IARG = 1
 END SUBROUTINE S1
 SUBROUTINE S2(RARG)
```

```
   RARG = 1.1
 END SUBROUTINE S2
 SUBROUTINE S3(LARG)
   LOGICAL LARG
   LARG = .TRUE.
 END SUBROUTINE S3
END

USE M
INTERFACE SS
  SUBROUTINE SS1(IARG,JARG)
  END SUBROUTINE
  MODULE PROCEDURE S1,S2,S3
END INTERFACE
CALL SS(II)             ! Calls subroutine S1 from M
CALL SS(I,J)            ! Calls subroutine SS1
END

SUBROUTINE SS1(IARG,JARG)
  IARG = 2
  JARG = 3
END SUBROUTINE
```

You can always reference a procedure through its specific interface. If a generic interface exists for a procedure, the procedure can also be referenced through the generic interface.

Within an interface body, if a dummy argument is intended to be a dummy procedure, it must have the **EXTERNAL** attribute or there must be an interface for the dummy argument.

Generic Interface Blocks

A generic interface block must specify a generic name, defined operator, or defined assignment in an **INTERFACE** statement. The generic name is a single name with which to reference all of the procedures specified in the interface block. It can be the same as any accessible generic name, or any of the procedure names in the interface block.

If two or more generic interfaces that are accessible in a scoping unit have the same local name, they are interpreted as a single generic interface.

Unambiguous Generic Procedure References

Whenever a generic procedure reference is made, only one specific procedure is invoked. The following rules ensure that a generic reference is unambiguous.

If two procedures in the same scoping unit both define assignment or both have the same defined operator and the same number of arguments, you

must specify a dummy argument that corresponds by position in the argument list to a dummy argument of the other that has a different type, kind type parameter, or rank.

Within a scoping unit, two procedures that have the same generic name must both be subroutines or both be functions. Also, at least one of them must have a nonoptional dummy argument that both:

1. Corresponds by position in the argument list to a dummy argument that is either not present in the argument list of the other subprogram, or is present with a different type, kind type parameter, or rank.
2. Corresponds by argument keyword to a dummy argument not present in the other argument list, or present with a different type, kind type parameter, or rank.

When an interface block extends an intrinsic procedure (see the next section), the above rules apply as if the intrinsic procedure consisted of a collection of specific procedures, one procedure for each allowed set of arguments.

Notes:

1. Dummy arguments of type **BYTE** are considered to have the same type as corresponding 1-byte dummy arguments of type **INTEGER(1)**, **LOGICAL(1)**, and character.
2. When the **-qintlog** compiler option is specified, dummy arguments of type integer and logical are considered to have the same type as corresponding dummy arguments of type integer and logical with the same kind type parameter.
3. If the dummy argument is only declared with the **EXTERNAL** attribute within an interface body, the dummy argument must be the only dummy argument corresponding by position to a procedure, and it must be the only dummy argument corresponding by argument keyword to a procedure.

Example of a Generic Interface Block

```
PROGRAM MAIN
INTERFACE A
  FUNCTION AI(X)
    INTEGER AI, X
  END FUNCTION AI
END INTERFACE
INTERFACE A
  FUNCTION AR(X)
    REAL AR, X
  END FUNCTION AR
END INTERFACE
INTERFACE FUNC
  FUNCTION FUNC1(I, EXT)      ! Here, EXT is a procedure
    INTEGER I
    EXTERNAL EXT
```

```
  END FUNCTION FUNC1
  FUNCTION FUNC2(EXT, I)
    INTEGER I
    REAL EXT                  ! Here, EXT is a variable
  END FUNCTION FUNC2
END INTERFACE
EXTERNAL MYFUNC
IRESULT=A(INTVAL)             ! Call to function AI
RRESULT=A(REALVAL)            ! Call to function AR
RESULT=FUNC(1,MYFUNC)         ! Call to function FUNC1
END PROGRAM MAIN
```

Extending Intrinsic Procedures with Generic Interface Blocks

A generic intrinsic procedure can be extended or redefined. An extended intrinsic procedure supplements the existing specific intrinsic procedures. A redefined intrinsic procedure replaces an existing specific intrinsic procedure.

When a generic name is the same as a generic intrinsic procedure name and the name has the **INTRINSIC** attribute (or appears in an intrinsic context), the generic interface extends the generic intrinsic procedure.

When a generic name is the same as a generic intrinsic procedure name and the name does not have the **INTRINSIC** attribute (nor appears in an intrinsic context), the generic interface can redefine the generic intrinsic procedure.

A generic interface name cannot be the same as a specific intrinsic procedure name if the name has the **INTRINSIC** attribute (or appears in an intrinsic context).

Example of Extending and Redefining Intrinsic Procedures

```
PROGRAM MAIN
INTRINSIC MAX
INTERFACE MAX               ! Extension to intrinsic MAX
  FUNCTION MAXCHAR(STRING)
    CHARACTER(50) STRING
  END FUNCTION MAXCHAR
END INTERFACE
INTERFACE ABS               ! Redefines generic ABS as
  FUNCTION MYABS(ARG)       !   ABS does not appear in
    REAL(8) MYABS, ARG      !   an INTRINSIC statement
  END FUNCTION MYABS
END INTERFACE
REAL(8) DARG, DANS
REAL(4) RANS
INTEGER IANS,IARG
CHARACTER(50) NAME
DANS = ABS(DARG)            ! Calls external MYABS
IANS = ABS(IARG)            ! Calls intrinsic IABS
```

```
DANS = DABS(DARG)           ! Calls intrinsic DABS
IANS = MAX(NAME)            ! Calls external MAXCHAR
RANS = MAX(1.0,2.0)         ! Calls intrinsic AMAX1
END PROGRAM MAIN
```

Defined Operators

A defined operator is a user-defined unary or binary operator, or an extended intrinsic operator (see "Extended Intrinsic and Defined Operations" on page 99). It must be defined by both a function and a generic interface block.

1. To define the unary operation *op* x_1:
   a. A function or entry must exist that specifies exactly one dummy argument, d_1.
   b. The *generic_spec* in an **INTERFACE** statement specifies **OPERATOR** *(op)*.
   c. The type of x_1 is the same as the type of the dummy argument d_1.
   d. The type parameters, if any, of x_1 must match those of d_1.
   e. If the function is not **ELEMENTAL**, the rank of x_1 (and its shape if it is an array) must match that of d_1.
2. To define the binary operation x_1 *op* x_2:
   a. The function is specified with a **FUNCTION** or **ENTRY** statement that specifies two dummy arguments, d_1 and d_2.
   b. The *generic_spec* in an **INTERFACE** block specifies **OPERATOR** *(op)*.
   c. The types of x_1 and x_2 are the same as those of the dummy arguments d_1 and d_2, respectively.
   d. The type parameters, if any, of x_1 and x_2 match those of d_1 and d_2, respectively.
   e. If the function is not **ELEMENTAL**, the ranks of x_1 and x_2 (and their shapes if either or both are arrays) match those of d_1 and d_2, respectively.
3. If *op* is an intrinsic operator, the types or ranks of either x_1 or x_2 are not those required for an intrinsic operation.
4. The *generic_spec* must not specify **OPERATOR** for functions with no arguments or for functions with more than two arguments.
5. Each argument must be nonoptional.
6. The arguments must be specified with **INTENT(IN)**.
7. Each function specified in the interface block cannot have a result of assumed character length.
8. If the operator specified is an intrinsic operator, the number of function arguments must be consistent with the intrinsic uses of that operator.
9. A given defined operator can, as with generic names, apply to more than one function, in which case it is generic just like generic procedure names. For intrinsic operator symbols, the generic properties include the intrinsic operations they represent.
10. The following rules apply only to extended intrinsic operations:

a. The type of one of the arguments can only be of type **BYTE** when the type of the other argument is of derived type.
b. When the **-qintlog** compiler option has been specified for non-character operations, and d_1 is numeric or logical, then d_2 must not be numeric or logical.
c. When the **-qctyplss** compiler option has been specified for non-character operations, if x_1 is numeric or logical and x_2 is a character constant, the intrinsic operation is performed.

Example of a Defined Operator

```
INTERFACE OPERATOR (.DETERMINANT.)
   FUNCTION IDETERMINANT (ARRAY)
    INTEGER, INTENT(IN), DIMENSION (:,:) :: ARRAY
     INTEGER IDETERMINANT
  END FUNCTION
 END INTERFACE
END
```

Defined Assignment

A defined assignment is treated as a reference to a subroutine, with the left-hand side as the first argument and the right-hand side enclosed in parentheses as the second argument.

1. To define the defined assignment $x_1 = x_2$:
   a. The subroutine is specified with a **SUBROUTINE** or **ENTRY** statement that specifies two dummy arguments, d_1 and d_2.
   b. The *generic_spec* of an interface block specifies **ASSIGNMENT (=)**.
   c. The types of x_1 and x_2 are the same as those of the dummy arguments d_1 and d_2, respectively.
   d. The type parameters, if any, of x_1 and x_2 match those of d_1 and d_2, respectively.
   e. If the subroutine is not **ELEMENTAL**, the ranks of x_1 and x_2 (and their shapes if either or both are arrays) must match those of d_1 and d_2, respectively.
2. **ASSIGNMENT** must only be used for subroutines with exactly two arguments.
3. Each argument must be nonoptional.
4. The first argument must have **INTENT(OUT)** or **INTENT(INOUT)**, and the second argument must have **INTENT(IN)**.
5. The types of the arguments must not be both numeric, both logical, or both character with the same kind parameter.

   The type of one of the arguments can only be of type **BYTE** when the type of the other argument is of derived type.

   When the **-qintlog** compiler option has been specified, and d_1 is numeric or logical, then d_2 must not be numeric or logical.

When the **-qctyplss** compiler option has been specified, if x_1 is numeric or logical and x_2 is a character constant, intrinsic assignment is performed.

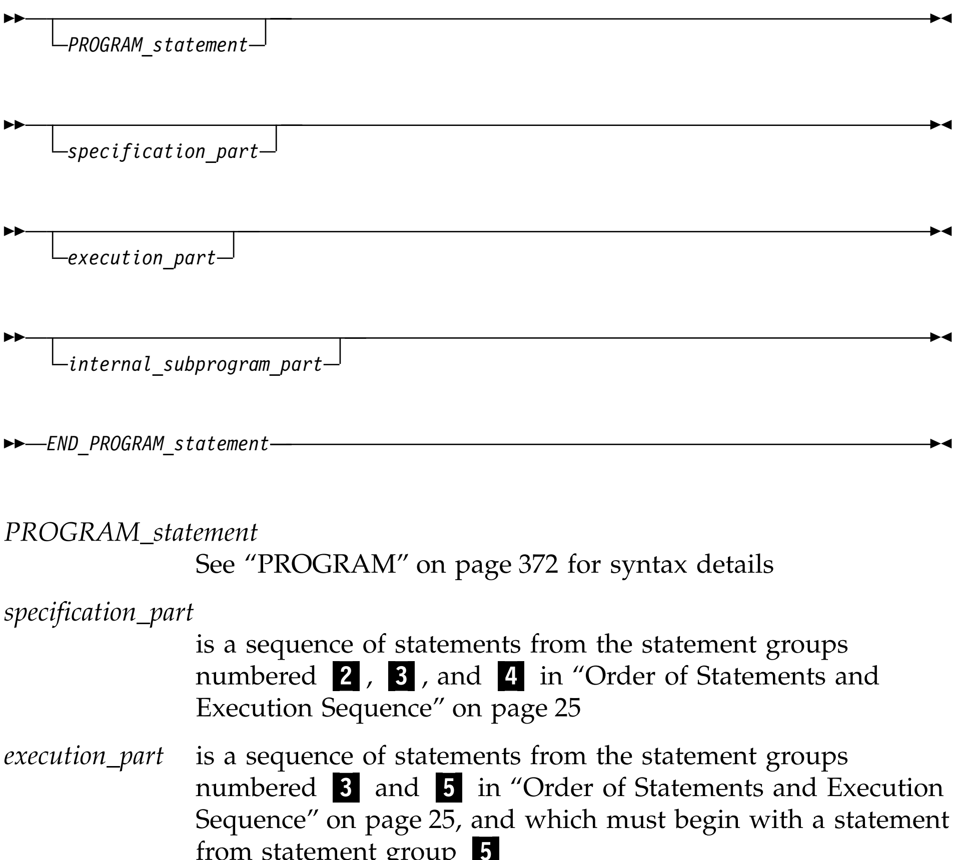
6. The **ASSIGNMENT** generic specification specifies that the assignment operation is extended or redefined if both sides of the equal sign are of the same derived type.

Example of Defined Assignment

```
INTERFACE ASSIGNMENT(=)
  SUBROUTINE BIT_TO_NUMERIC (N,B)
    INTEGER, INTENT(OUT) :: N
    LOGICAL, INTENT(IN), DIMENSION(:) :: B
  END SUBROUTINE
END INTERFACE
```

Main Program

A main program is the program unit that receives control from the system when the executable program is invoked at run time.



PROGRAM_statement
: See “PROGRAM” on page 372 for syntax details

specification_part
: is a sequence of statements from the statement groups numbered **2**, **3**, and **4** in “Order of Statements and Execution Sequence” on page 25

execution_part
: is a sequence of statements from the statement groups numbered **3** and **5** in “Order of Statements and Execution Sequence” on page 25, and which must begin with a statement from statement group **5**

internal_subprogram_part
: See "Internal Procedures" on page 141 for details

END_PROGRAM_statement
: See "END" on page 296 for syntax details

A main program cannot contain an **ENTRY** statement, nor can it specify an automatic object.

A **RETURN** statement can appear in a main program. The execution of a **RETURN** statement has the same effect as the execution of an **END** statement.

A main program cannot reference itself, directly or indirectly.

Modules

A module contains specifications and definitions that can be accessed from other program units. These definitions include data object definitions, namelist groups, derived-type definitions, procedure interface blocks and procedure definitions.

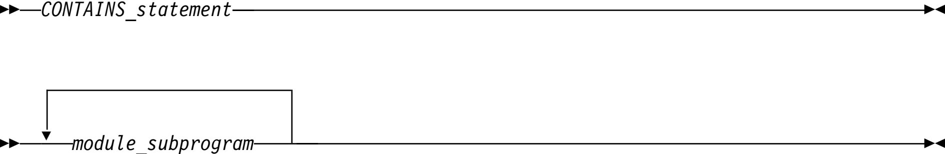
Fortran 90 modules define global data, which, like **COMMON** data, is shared across threads and is therefore thread-unsafe. To make an application thread-safe, you must put the module data into a named **COMMON** block within the module, and declare that named **COMMON** block as **THREADLOCAL**. See "COMMON" on page 267 and "THREADLOCAL" on page 507 for more information.

```
>>──MODULE_statement─────────────────────────────────────────────><

>>──┬──────────────────────┬─────────────────────────────────────><
    └─specification_part───┘

>>──┬──────────────────────────┬─────────────────────────────────><
    └─module_subprogram_part───┘

>>──END_MODULE_statement─────────────────────────────────────────><
```

MODULE_statement
: See "MODULE" on page 350 for syntax details

specification_part
: is a sequence of statements from the statement groups numbered **2**, **3**, and **4** in "Order of Statements and Execution Sequence" on page 25

module_subprogram_part:



CONTAINS_statement
: See “CONTAINS” on page 274 for syntax details

END_MODULE_statement
: See “END” on page 296 for syntax details

A module subprogram is contained in a module but is not an internal subprogram. Module subprograms must follow a **CONTAINS** statement, and can contain internal procedures. A module procedure is defined by a module subprogram or an entry in a module subprogram.

Executable statements within a module can only be specified in module subprograms.

The declaration of a module function name of type character cannot have an asterisk as a length specification.

specification_part cannot contain statement function statements, **ENTRY** statements, or **FORMAT** statements, although these statements can appear in the specification part of a module subprogram.

Automatic objects and objects with the **AUTOMATIC** attribute cannot appear in the scope of a module.

Integer pointers cannot appear in *specification_part* if the pointee specifies a dimension declarator with nonconstant bounds.

An accessible module procedure can be invoked by another subprogram in the module or by any scoping unit outside the module through use association (that is, by using the **USE** statement). See “USE” on page 408 for details.

All objects in the scope of a module retain their association status, allocation status, definition status, and value when any procedure that accesses the module through use association executes a **RETURN** or **END** statement. See point 4 under “Events Causing Undefinition” on page 55 for more information.

A module is a host to any module procedures or derived-type definitions it contains, which can access entities in the scope of the module through host association.

A module procedure can be used as an actual argument associated with a dummy procedure argument.

The name of a module procedure is local to the scope of the module and cannot be the same as the name of any entity in the module, except for a common block name.

Migration Tips:
- Eliminate common blocks and **INCLUDE** directives
- Use modules to hold global data and procedures to ensure consistency of definitions

FORTRAN 77 source:

```
COMMON /BLOCK/A, B, C, NAME, NUMBER
REAL A, B, C
A = 3
CALL CALLUP(D)
PRINT *, NAME, NUMBER
END
SUBROUTINE CALLUP (PARM)
  COMMON /BLOCK/A, B, C, NAME, NUMBER
  REAL A, B, C
  ...
  NAME = 3
  NUMBER = 4
END
```

Fortran 90 or Fortran 95 source:

```
MODULE FUNCS
  REAL A, B, C            ! Common block no longer needed
  INTEGER NAME, NUMBER    ! Global data
  CONTAINS
     SUBROUTINE CALLUP (PARM)
       ...
       NAME = 3
       NUMBER = 4
     END SUBROUTINE
END MODULE FUNCS
PROGRAM MAIN
USE FUNCS
A = 3
CALL CALLUP(D)
PRINT *, NAME, NUMBER
END
```

Example of a Module

```
MODULE M
  INTEGER SOME_DATA
  CONTAINS
    SUBROUTINE SUB()                     ! Module subprogram
      INTEGER STMTFNC
      STMTFNC(I) = I + 1
      SOME_DATA = STMTFNC(5) + INNER(3)
      CONTAINS
        INTEGER FUNCTION INNER(IARG)     ! Internal subprogram
          INNER = IARG * 2
        END FUNCTION
    END SUBROUTINE SUB
END MODULE
PROGRAM MAIN
  USE M                                  ! Main program accesses
  CALL SUB()                             !   module M
END PROGRAM
```

Block Data Program Unit

A block data program unit provides initial values for objects in named common blocks.

```
►►──BLOCK_DATA_statement─────────────────────────────────────────────►◄

►►──┬────────────────────┬───────────────────────────────────────────►◄
    └─specification_part─┘

►►──END_BLOCK_DATA_statement─────────────────────────────────────────►◄
```

BLOCK_DATA_statement
: See "BLOCK DATA" on page 252 for syntax details

specification_part
: is a sequence of statements from the statement groups numbered **2**, **3**, and **4** in "Order of Statements and Execution Sequence" on page 25

END_BLOCK_DATA_statement
: See "END" on page 296 for syntax details

In *specification_part*, you can specify type declaration, **USE**, **IMPLICIT**, **COMMON**, **DATA**, **EQUIVALENCE**, and integer **POINTER** statements, derived-type definitions, and the allowable attribute specification statements. The only attributes that can be specified include **PARAMETER**, **DIMENSION**, **INTRINSIC**, **POINTER**, **SAVE**, and **TARGET**.

You can have more than one block data program unit in an executable program, but only one can be unnamed. You can also initialize multiple named common blocks in a block data program unit.

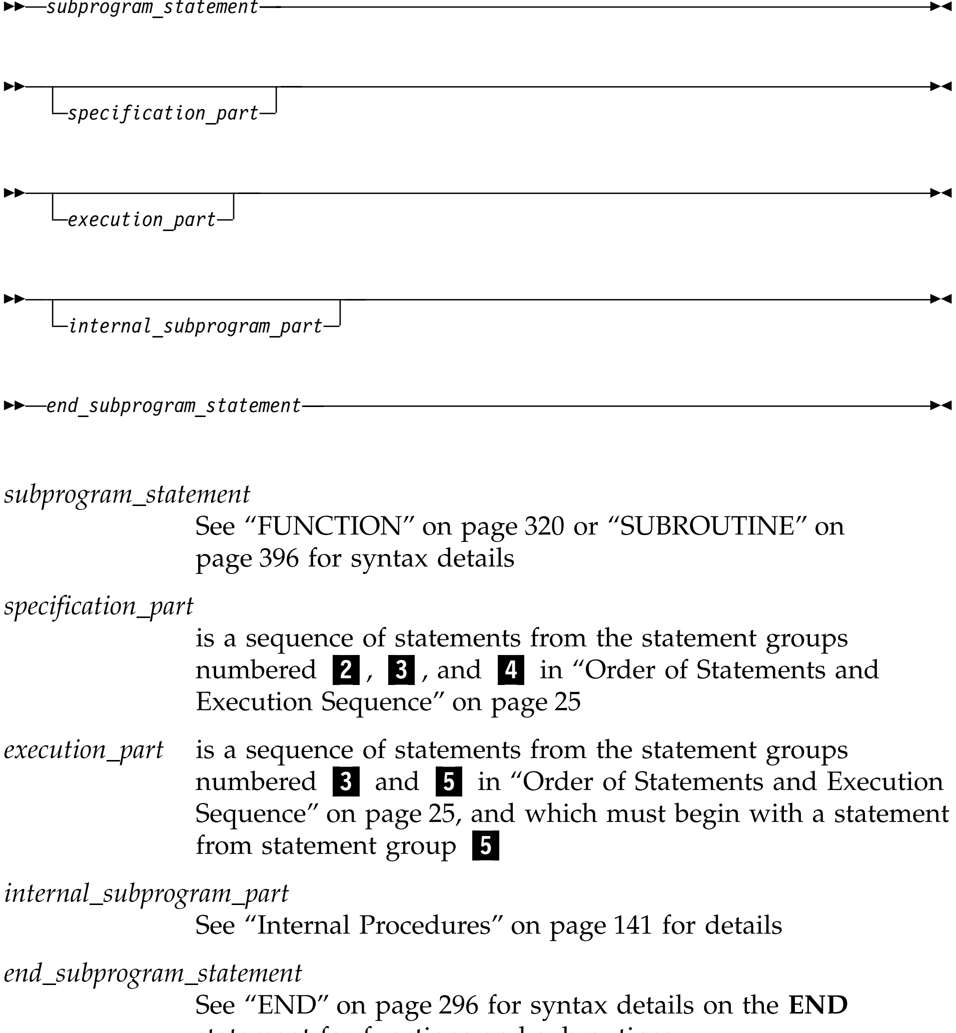
Restrictions on common blocks in block data program units are:
- All items in a named common block must appear in the **COMMON** statement, even if they are not all initialized.
- The same named common block must not be referenced in two different block data program units.
- Only nonpointer objects in named common blocks can be initialized in block data program units.
- Objects in blank common blocks cannot be initialized.

Example of a Block Data Program Unit

```
PROGRAM MAIN
  COMMON /L3/ C, X(10)
  COMMON /L4/ Y(5)
END PROGRAM
BLOCK DATA BDATA
  COMMON /L3/ C, X(10)
  DATA C, X /1.0, 10*2.0/   ! Initializing common block L3
END BLOCK DATA

BLOCK DATA                  ! An unnamed block data program unit
  PARAMETER (Z=10)
  DIMENSION Y(5)
  COMMON /L4/ Y
  DATA Y /5*Z/
END BLOCK DATA
```

Function and Subroutine Subprograms

A subprogram is either a function or a subroutine, and is either an internal, external, or module subprogram. You can also specify a function in a statement function statement. An external subprogram is a program unit.



subprogram_statement
: See "FUNCTION" on page 320 or "SUBROUTINE" on page 396 for syntax details

specification_part
: is a sequence of statements from the statement groups numbered **2**, **3**, and **4** in "Order of Statements and Execution Sequence" on page 25

execution_part
: is a sequence of statements from the statement groups numbered **3** and **5** in "Order of Statements and Execution Sequence" on page 25, and which must begin with a statement from statement group **5**

internal_subprogram_part
: See "Internal Procedures" on page 141 for details

end_subprogram_statement
: See "END" on page 296 for syntax details on the **END** statement for functions and subroutines

An internal subprogram is declared *after* the **CONTAINS** statement in the main program, a module subprogram, or an external subprogram, but *before* the **END** statement of the host program. The name of an internal subprogram must not be defined in the specification section in the host scoping unit.

An external procedure has global scope with respect to the executable program. In the calling program unit, you can specify the interface to an external procedure in an interface block or you can define the external procedure name with the **EXTERNAL** attribute.

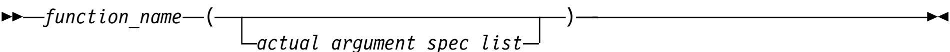
A subprogram can contain any statement except **PROGRAM**, **BLOCK DATA** and **MODULE** statements. An internal subprogram cannot contain an **ENTRY** statement or an internal subprogram.

Procedure References

There are two types of procedure references:
- A subroutine is invoked by execution of a **CALL** statement (see "CALL" on page 257 for details) or defined assignment statement.
- A function is invoked during evaluation of a function reference or defined operation.

Function Reference

A function reference is used as a primary in an expression:



Executing a function reference results in the following order of events:
1. Actual arguments that are expressions are evaluated.
2. Actual arguments are associated with their corresponding dummy arguments.
3. Control transfers to the specified function.
4. The function is executed.
5. The value (or status or target, for pointer functions) of the function result variable is available to the referencing expression.

Execution of a function reference must not alter the value of any other data item within the statement in which the function reference appears. Invocation of a function reference in the logical expression of a logical **IF** statement or **WHERE** statement can affect entities in the statement that is executed when the value of the expression is true.

The argument list built-in functions **%VAL** and **%REF** are supplied to aid interlanguage calls by allowing arguments to be passed by value and by reference, respectively. They can be specified in non-Fortran procedure references and in a subprogram statement in an interface body. (See "%VAL and %REF" on page 165.) See "Examples" on page 393 and "Examples" on page 323 for examples of function references.

Examples of Subprograms and Procedure References

```
PROGRAM MAIN
REAL QUAD,X2,X1,X0,A,C3
QUAD=0; A=X1*X2
X2 = 2.0
X1 = SIN(4.5)                   ! Reference to intrinsic function
X0 = 1.0
CALL Q(X2,X1,X0,QUAD)           ! Reference to external subroutine
```

```
C3 = CUBE()                       ! Reference to internal function
CONTAINS
  REAL FUNCTION CUBE()            ! Internal function
    CUBE = A**3
  END FUNCTION CUBE
END
SUBROUTINE Q(A,B,C,QUAD)          ! External subroutine
  REAL A,B,C,QUAD
  QUAD = (-B + SQRT(B**2-4*A*C)) / (2*A)
END SUBROUTINE Q
```

Intrinsic Procedures

An intrinsic procedure is a procedure already defined by XL Fortran. See "Chapter 12. Intrinsic Procedures" on page 519 for details.

You can reference some intrinsic procedures by a generic name, some by a specific name, and some by both:

A generic intrinsic function
: does not require a specific argument type and usually produces a result of the same type as that of the argument, with some exceptions. Generic names simplify references to intrinsic procedures because the same procedure name can be used with more than one type of argument; the type and kind type parameter of the arguments determine which specific function is used.

A specific intrinsic function
: requires a specific argument type and produces a result of a specific type.

  A specific intrinsic function name can be passed as an actual argument. If a specific intrinsic function has the same name as a generic intrinsic function, the specific name is referenced. All references to a dummy procedure that are associated with a specific intrinsic procedure must use arguments that are consistent with the interface of the intrinsic procedure.

Whether or not you can pass the name of an intrinsic procedure as an argument depends on the procedure. You can use the specific name of an intrinsic procedure that has been specified with the **INTRINSIC** attribute as an actual argument in a procedure reference.

- An **IMPLICIT** statement does not change the type of an intrinsic function.
- If an intrinsic name is specified with the **INTRINSIC** attribute, the name is always recognized as an intrinsic procedure.

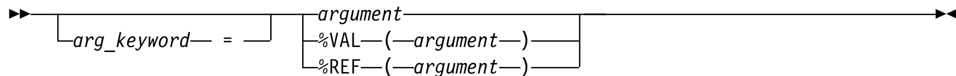
Conflicts Between Intrinsic Procedure Names and Other Names

Because intrinsic procedure names are recognized, when a data object is declared with the same name as an intrinsic procedure, the intrinsic procedure is inaccessible.

A generic interface block can extend or redefine a generic intrinsic function, as described in “Interface Blocks” on page 144. If the function already has the **INTRINSIC** attribute, it is extended; otherwise, it can be redefined.

Arguments

Actual Argument Specification



arg_keyword
: is a dummy argument name in the explicit interface of the procedure being invoked

argument
: is an actual argument

%VAL, %REF
: specifies the passing method. See “%VAL and %REF” on page 165 for more information.

An actual argument appears in the argument list of a procedure reference. An actual argument in a procedure reference can be one of the following:

- An expression
- A variable
- A procedure name
- An alternate return specifier (if the actual argument is in a **CALL** statement), having the form **stmt_label*, where *stmt_label* is the statement label of a branch target statement in the same scoping unit as the **CALL** statement.

An actual argument specified in a statement function reference must be a scalar object.

A procedure name cannot be the name of an internal procedure, statement function, or the generic name of a procedure, unless it is also a specific name.

The rules and restrictions for referencing a procedure described in “Procedure References” on page 159. In addition, you cannot use a non-intrinsic elemental procedure as an actual argument.

Argument Keywords

Argument keywords allow you to specify actual arguments in a different order than the dummy arguments. With argument keywords, any actual arguments that correspond to optional dummy arguments can be omitted; that is, dummy arguments that merely serve as placeholders are not necessary.

Each argument keyword must be the name of a dummy argument in the explicit interface of the procedure being referenced. An argument keyword must not appear in an argument list of a procedure that has an implicit interface.

In the argument list, if an actual argument is specified with an argument keyword, the subsequent actual arguments in the list must also be specified with argument keywords.

An argument keyword cannot be specified for label parameters. Label parameters must appear before referencing the argument keywords in that procedure reference.

Example of Argument Keywords:

```
INTEGER MYARRAY(1:10)
INTERFACE
  SUBROUTINE SORT(ARRAY, DESCENDING, ARRAY_SIZE)
    INTEGER ARRAY_SIZE, ARRAY(ARRAY_SIZE)
    LOGICAL, OPTIONAL :: DESCENDING
  END SUBROUTINE
END INTERFACE
CALL SORT(MYARRAY, ARRAY_SIZE=10)  ! No actual argument corresponds to the
                                   ! optional dummy argument DESCENDING
END
SUBROUTINE SORT(ARRAY, DESCENDING, ARRAY_SIZE)
  INTEGER ARRAY_SIZE, ARRAY(ARRAY_SIZE)
  LOGICAL, OPTIONAL :: DESCENDING
  IF (PRESENT(DESCENDING)) THEN
            .
            .
            .
END SUBROUTINE
```

Dummy Arguments

```
►►──┬─dummy_arg_name──────────┬─────────────────────────────────────────►◄
    ├─%VAL──(──dummy_arg_name──)─┤
    └─%REF──(──dummy_arg_name──)─┘
```

A dummy argument is specified in a Statement Function statement, **FUNCTION** statement, **SUBROUTINE** statement, or **ENTRY** statement. Dummy arguments in statement functions, function subprograms, interface bodies, and subroutine subprograms indicate the types of actual arguments and whether each argument is a scalar value, array, procedure, or statement label. A dummy argument in an external, module, or internal subprogram definition, or in an interface body, is classified as one of the following:
- A variable name
- A procedure name
- An asterisk (in subroutines only, to indicate an alternate return point)

%VAL or **%REF** can only be specified for a dummy argument in a **FUNCTION** or **SUBROUTINE** statement in an interface block. The interface must be for a non-Fortran procedure interface. If **%VAL** or **%REF** appears in an interface block for an external procedure, this passing method is implied for each reference to that procedure. If an actual argument in an external procedure reference specifies **%VAL** or **%REF**, the same passing method must be specified in the interface block for the corresponding dummy argument. See “%VAL and %REF” on page 165 for more details.

A dummy argument in a statement function definition is classified as a variable name.

A given name can appear only once in a dummy argument list.

The name of a variable that appears as a dummy argument in a statement function statement has a scope of the statement in which it appears. It has the type that it would have if it were the name of a variable in the scoping unit that includes the statement function. It cannot have the same name as an accessible array.

Argument Association

Actual arguments are associated with dummy arguments when a function or subroutine is referenced. In a procedure reference, the actual argument list identifies the correspondence between the actual arguments provided in the list and the dummy arguments of the subprogram.

When there is no argument keyword, an actual argument is associated with the dummy argument that occupies the corresponding position in the dummy argument list. The first actual argument becomes associated with the first dummy argument, the second actual argument with the second dummy argument, and so forth. Each actual argument must be associated with a dummy argument.

When a keyword is present, the actual argument is associated with the dummy argument whose name is the same as the argument keyword. In the scoping unit that contains the procedure reference, the names of the dummy arguments must exist in an accessible explicit interface.

Argument association within a subprogram terminates upon execution of a **RETURN** or **END** statement in the subprogram. There is no retention of argument association between one reference of a subprogram and the next reference of the subprogram, unless the **persistent** suboption of the **-qxlf77** compiler option is specified and the subprogram contains at least one entry procedure.

Except when **%VAL** is used, the subprogram reserves no storage for the dummy argument. It uses the corresponding actual argument for calculations. Therefore, the value of the actual argument changes when the dummy argument changes. If the corresponding actual argument is an expression or an array section with vector subscripts, the calling procedure reserves storage for the actual argument, and the subprogram must not define, redefine, or undefine the dummy argument.

If the actual argument is specified with **%VAL**, the subprogram does not have access to the storage area of the actual argument.

Actual arguments must agree in type and type parameters with their corresponding dummy arguments (and in shape if the dummy arguments are pointers or assumed-shape), except for two cases: a subroutine name has no type and must be associated with a dummy procedure name that is a subroutine, and an alternate return specifier has no type and must be associated with an asterisk.

Argument association can be carried through more than one level of procedure reference.

If a subprogram reference causes a dummy argument in the referenced subprogram to become associated with another dummy argument in the referenced subprogram, neither dummy argument can become defined, redefined, or undefined during that subprogram. For example, if a subroutine definition is:

```
SUBROUTINE XYZ (A,B)
```

and it is referenced by:

```
CALL XYZ (C,C)
```

the dummy arguments A and B each become associated with the same actual argument C and, therefore, with each other. Neither A nor B can be defined, redefined, or undefined during the execution of subroutine XYZ or by any procedures referenced by XYZ.

If a dummy argument becomes associated with an entity in a common block or an entity accessible through use or host association, the value of the entity must only be altered through the use of the dummy argument name, while the entity is associated with the dummy argument. If any part of a data object is defined through a dummy argument, the data object can be referenced only through that dummy argument, either before or after the definition occurs. These restrictions also apply to pointer targets.

If you have programs that do not conform to these restrictions, using the compiler option **-qalias=nostd** may be appropriate. See "-qalias Option" in the *User's Guide* for details.

%VAL and %REF

To call subprograms written in languages other than Fortran (for example, user-written C programs, or AIX operating system routines), the actual arguments may need to be passed by a method different from the default method used by XL Fortran. The default method passes the address of the actual argument and, if it is of type character, the length. (Use the **-qnullterm** compiler option to ensure that scalar character initialization expressions are passed with terminating null strings. See "-qnullterm Option" in the *User's Guide* for details.)

The default passing method can be changed by using the **%VAL** and **%REF** built-in functions in the argument list of a **CALL** statement or function reference, or with the dummy arguments in interface bodies. These built-in functions specify the way an actual argument is passed to the external subprogram.

%VAL and **%REF** built-in functions cannot be used in the argument lists of Fortran procedure references, nor can they be used with alternate return specifiers.

The argument list built-in functions are:

%VAL This built-in function can be used with actual arguments that are **CHARACTER(1)**, logical, integer, real, complex expressions, or sequence derived type. Objects of derived type cannot contain character structure components whose lengths are greater than 1 byte, or arrays.

%VAL cannot be used with actual arguments that are arrays, procedure names, or character expressions of length greater than 1 byte.

%VAL causes the actual argument to be passed as 32-bit or 64-bit intermediate values. If the actual argument is of type real or complex, it is passed as one or more 64-bit intermediate values. If the actual argument is of integer, logical, or sequence derived type, it is passed as one or more 32-bit intermediate values. An integer actual argument shorter than 32 bits is sign-extended to a 32-bit value, while a logical actual argument shorter than 32 bits is padded with zeros to a 32-bit value.

Byte named constants and variables are passed as if they were **INTEGER(1)**. If the actual argument is a **CHARACTER(1)**, it is padded on the left with zeros to a 32-bit value, regardless of whether the **-qctyplss** compiler option is specified.

%REF This built-in function causes the actual argument to be passed by reference; that is, only the address of the actual argument is passed. Unlike the default passing method, **%REF** does not pass the length of a character argument. If such a character argument is being passed to a C routine, the string must be terminated with a null character (for example, using the **-qnullterm** option) so that the C routine can determine the length of the string.

Examples of %VAL and %REF

```
EXTERNAL FUNC
CALL RIGHT2(%REF(FUNC))      ! procedure name passed by reference
COMPLEX XVAR
CALL RIGHT3(%VAL(XVAR))      ! complex argument passed by value

IVARB=6
CALL TPROG(%VAL(IVARB))      ! integer argument passed by value
```

See "Interlanguage Calls" in the *User's Guide* for more information.

Intent of Dummy Arguments

With the **INTENT** attribute, you can explicitly specify the intended use of a dummy argument. Use of this attribute may improve optimization of the program's calling procedure when an explicit interface exists. Also, the explicitness of argument intent may provide more opportunities for error checking. See "INTENT" on page 342 for syntax details.

The following table outlines XL Fortran's passing method for internal procedures (not including assumed-shape dummy arguments and pointer dummy arguments):

Table 5. Passing Method and Intent

Argument Type	Intent(IN)	Intent(OUT)	Intent(INOUT)	No Intent
Non-CHARACTER Scalar	**VALUE**	default	default	default
CHARACTER*1 Scalar	**VALUE**	**REFERENCE**	**REFERENCE**	**REFERENCE**
CHARACTER*n Scalar	**REFERENCE**	**REFERENCE**	**REFERENCE**	**REFERENCE**
CHARACTER*(*) Scalar	default	default	default	default
Derived Type [1] Scalar	**VALUE**	default	default	default
Derived Type [2] Scalar	default	default	default	default
Non-CHARACTER Array	default	default	default	default
CHARACTER*1 Array	**REFERENCE**	**REFERENCE**	**REFERENCE**	**REFERENCE**
CHARACTER*n Array	**REFERENCE**	**REFERENCE**	**REFERENCE**	**REFERENCE**
CHARACTER*(*) Array	default	default	default	default
Derived Type [3] Array	default	default	default	default

Optional Dummy Arguments

The **OPTIONAL** attribute specifies that a dummy argument need not be associated with an actual argument in a reference to a procedure. Some advantages of the **OPTIONAL** attribute include:

- The use of optional dummy arguments to override default behavior. For an example, see "Example of Argument Keywords" on page 162.
- Additional flexibility in procedure references. For example, a procedure could include optional arguments for error handlers or return codes, but you can select which procedure references would supply the corresponding actual arguments.

See "OPTIONAL" on page 361 for details about syntax and rules.

1. A data object of derived type with no array components or CHARACTER*n components, (where $n > 1$).
2. A data object of derived type with array components or CHARACTER*n components, (where $n > 1$).
3. A data object of derived type with components of any type, size and rank.

Restrictions on Optional Dummy Arguments Not Present

A dummy argument is present in an instance of a subprogram if it is associated with an actual argument, and the actual argument is either a dummy argument that is not optional in the invoking subprogram or a dummy argument that is not present in the invoking subprogram. A dummy argument that is not optional must be present.

An optional dummy argument that is not present must conform to the following rules:

- If it is a dummy data object, it must not be referenced or defined. If the dummy data object is of a type for which default initialization can be specified, the initialization has no effect.
- If it is a dummy procedure, it must not be invoked.
- It must not be supplied as an actual argument that corresponds to a nonoptional dummy argument, except as the argument of the **PRESENT** intrinsic function.
- A subobject of an optional dummy argument that is not present must not be supplied as an actual argument that corresponds to an optional dummy argument.
- If the optional dummy argument that is not present is an array, it must not be supplied as an actual argument to an elemental procedure unless an array of the same rank is supplied as an actual argument that corresponds to a nonoptional dummy argument of that elemental procedure.
- If the optional dummy argument that is not present is a pointer, it must not be supplied as an actual argument that corresponds to a nonpointer dummy argument, except as the argument of the **PRESENT** intrinsic function.

Length of Character Arguments

If the length of a character dummy argument is a nonconstant specification expression, the object is a dummy argument with a run-time length. If an object that is not a dummy argument has a run-time length, it is an automatic object. See “Automatic Objects” on page 29 for details.

If a dummy argument has a length specifier of an asterisk in parentheses, the length of the dummy argument is “inherited” from the actual argument. The length is inherited because it is specified outside the program unit containing the dummy argument. If the associated actual argument is an array name, the length inherited by the dummy argument is the length of an array element in the associated actual argument array. **%REF** cannot be specified for a character dummy argument with inherited length.

Variables as Dummy Arguments

A dummy argument that is a variable must be associated with an actual argument that is a variable with the same type and kind type parameter.

If the actual argument is scalar, the corresponding dummy argument must be scalar, unless the actual argument is an element of an array that is not an assumed-shape or pointer array (or a substring of such an element). If the procedure is referenced by a generic name or as a defined operator or defined assignment, the ranks of the actual arguments and corresponding dummy arguments must agree. A scalar dummy argument can be associated only with a scalar actual argument.

The following apply to dummy arguments used in elemental subprograms:

- All dummy arguments must be scalar, and cannot have the **POINTER** attribute.
- A dummy argument, or a subobject thereof, cannot be used in a specification expression, except if it is used as an argument to the **BIT_SIZE**, **KIND**, or **LEN** intrinsic functions, or as an argument to one of the numeric inquiry intrinsic functions, see "Chapter 12. Intrinsic Procedures" on page 519.
- A dummy argument cannot be an asterisk.
- A dummy argument cannot be a dummy procedure.

If a scalar dummy argument is of type character, its length must be less than or equal to the length of the actual argument. The dummy argument is associated with the leftmost characters of the actual argument. If the character dummy argument is an array, the length restriction applies to the entire array rather than each array element. That is, the lengths of associated array elements can vary, although the whole dummy argument array cannot be longer than the whole actual argument array.

If the dummy argument is an assumed-shape array, the actual argument must not be an assumed-size array or a scalar (including a designator for an array element or an array element substring).

If the dummy argument is an explicit-shape or assumed-size array, and if the actual argument is a noncharacter array, the size of the dummy argument must not exceed the size of the actual argument array. Each actual array element is associated with the corresponding dummy array element. If the actual argument is a noncharacter array element with a subscript value of `as`, the size of the dummy argument array must not exceed the size of the actual argument array `+ 1 - as`. The dummy argument array element with a subscript value of `ds` becomes associated with the actual argument array element that has a subscript value of `as + ds - 1`.

If an actual argument is a character array, character array element, or character substring, and begins at a character storage unit `acu` of an array, character storage unit `dcu` of an associated dummy argument array becomes associated with character storage unit `acu+dcu-1` of the actual array argument.

You can define a dummy argument that is a variable name within a subprogram if the associated actual argument is a variable. You must not redefine a dummy argument that is a variable name within a subprogram if the associated actual argument is not definable.

If the actual argument is an array section with a vector subscript, the associated dummy argument cannot be defined.

If a nonpointer dummy argument is associated with a pointer actual argument, the actual argument must be currently associated with a target, to which the dummy argument becomes argument associated. Any restrictions on the passing method apply to the target of the actual argument.

If the dummy argument is neither a target nor a pointer, any pointers associated with the actual argument do not become associated with the corresponding dummy argument on invocation of the procedure.

If both the dummy and actual arguments are targets, with the dummy argument being a scalar or an assumed-shape array (and the actual argument is not an array section with a vector subscript):
1. Any pointers associated with the actual argument become associated with the corresponding dummy argument on invocation of the procedure.
2. When execution of the procedure completes, any pointers associated with the dummy argument remain associated with the actual argument.

If both the dummy and actual arguments are targets, with the dummy argument being either an explicit-shape array or an assumed-size array, while the actual argument is not an array section with a vector subscript:
1. Whether any pointers associated with the actual argument become associated with the corresponding dummy argument on invocation of the procedure is processor dependent.
2. When execution of the procedure completes, whether any pointers associated with the dummy argument remain associated with the actual argument is processor dependent.

If the dummy argument is a target and the corresponding actual argument is not a target or is an array section with a vector subscript, any pointers associated with the dummy argument become undefined when execution of the procedure completes.

Pointers as Dummy Arguments

If a dummy argument is a pointer, the actual argument must be a pointer and their types, type parameters, and ranks must match. The actual argument reference is to the pointer itself, not to its target. When the procedure is invoked:

- The dummy argument acquires the pointer association status of the actual argument.
- If the actual argument is associated, the dummy argument is associated with the same target.

The association status can change during execution of the procedure. When the procedure finishes executing, the dummy argument's association status becomes undefined, if it is associated.

The passing method must be by reference; that is, **%VAL** must not be specified for the pointer actual argument.

Procedures as Dummy Arguments

A dummy argument that is identified as a procedure is called a dummy procedure. It can only be associated with an actual argument that is a specific intrinsic procedure, module procedure, external procedure, or another dummy procedure. See "Chapter 12. Intrinsic Procedures" on page 519 for details on which intrinsic procedures can be passed as actual arguments.

The dummy procedure and corresponding actual argument must both be functions or both be subroutines. Dummy arguments of the actual procedure argument must match those of the dummy procedure argument. If they are functions, they must match in type, type parameters, rank, shape (if they are nonpointer arrays), and whether they are pointers. If the length of a function result is assumed, this is a characteristic of the result. If the function result specifies a type parameter or array bound that is not a constant expression, the dependence on the entities in the expression is a characteristic of the result.

Dummy procedures that are subroutines are treated as if they have a type that is different from the intrinsic data types, derived types, and alternate return specifiers. Such dummy arguments only match actual arguments that are subroutines or dummy procedures.

Internal subprograms cannot be associated with a dummy procedure argument. The rules and restrictions for referencing a procedure described in "Procedure References" on page 159. In addition, you cannot use a non-intrinsic elemental procedure as an actual argument.

Examples of Procedures as Dummy Arguments

```
PROGRAM MYPROG
INTERFACE
  SUBROUTINE SUB (ARG1)
    EXTERNAL ARG1
    INTEGER ARG1
  END SUBROUTINE SUB
END INTERFACE
```

```
EXTERNAL IFUNC, RFUNC
REAL RFUNC

CALL SUB (IFUNC)    ! Valid reference
CALL SUB (RFUNC)    ! Invalid reference
!
! The first reference to SUB is valid because IFUNC becomes an
! implicitly declared integer, which then matches the explicit
! interface. The second reference is invalid because RFUNC is
! explicitly declared real, which does not match the explicit
! interface.
END PROGRAM
```

```
SUBROUTINE ROOTS
  EXTERNAL NEG
  X = QUAD(A,B,C,NEG)
  RETURN
END
FUNCTION QUAD(A,B,C,FUNCT)
  INTEGER FUNCT
  VAL = FUNCT(A,B,C)
  RETURN
END

FUNCTION NEG(A,B,C)
  RETURN
END
```

Asterisks as Dummy Arguments

A dummy argument that is an asterisk can only appear in the dummy argument list of a **SUBROUTINE** statement or an **ENTRY** statement in a subroutine subprogram. The corresponding actual argument must be an alternate return specifier, which indicates the statement label of a branch target statement in the same scope as the **CALL** statement, to which control is returned.

Example of an Alternate Return Specifier

```
   CALL SUB(*10)
   STOP                 ! STOP is never executed
10 PRINT *, 'RETURN 1'
   CONTAINS
     SUBROUTINE SUB(*)

   ⋮

       RETURN 1         ! Control returns to statement with label 10
     END SUBROUTINE
   END
```

Resolution of Procedure References

The subprogram name in a procedure reference is either established to be generic, established to be only specific, or not established.

A subprogram name is established to be generic in a scoping unit if one or more of the following is true:
- The scoping unit has an interface block with that name.
- The name of the subprogram is the same as the name of a generic intrinsic procedure that is specified in the scoping unit with the **INTRINSIC** attribute.
- The scoping unit accesses the generic name from a module through use association.
- There are no declarations of the subprogram name in the scoping unit, but the name is established to be generic in the host scoping unit.

A subprogram name is established to be only specific in a scoping unit when it has not been established to be generic and one of the following is true:
- An interface body in the scoping unit has the same name.
- There is a statement function, module procedure, or an internal subprogram in the scoping unit that has the same name.
- The name of the subprogram is the same as the name of a specific intrinsic procedure that is specified with the **INTRINSIC** attribute in the scoping unit.
- The scoping unit contains an **EXTERNAL** statement with the subprogram name.
- The scoping unit accesses the specific name from a module through use association.
- There are no declarations of the subprogram name in the scoping unit, but the name is established to be specific in the host scoping unit.

If a subprogram name is not established to be either generic nor specific, it is not established.

Rules for Resolving Procedure References to Names

The following rules are used to resolve a procedure reference to a name established to be generic:

1. If there is an interface block with that name in the scoping unit or accessible through use association, and the reference is consistent with one of the specific interfaces of that interface block, the reference is to the specific procedure associated with the specific interface.
2. If Rule 1 does not apply, the reference is to an intrinsic procedure if the procedure name in the scoping unit is specified with the **INTRINSIC** attribute or accesses a module entity whose name is specified with the **INTRINSIC** attribute, and the reference is consistent with the interface of that intrinsic procedure.
3. If neither Rule 1 nor Rule 2 applies, but the name is established to be generic in the host scoping unit, the name is resolved by applying Rule 1 and Rule 2 to the host scoping unit. For this rule to apply, there must be

agreement between the host scoping unit and the scoping unit of which the name is either a function or a subroutine.

4. If Rule 1, Rule 2 and Rule 3 do not apply, the reference must be to the generic intrinsic procedure with that name.

The following rules are used to resolve a procedure reference to a name established to be only specific:

1. If the scoping unit is a subprogram, and it contains either an interface body with that name or the name has the **EXTERNAL** attribute, and if the name is a dummy argument of that subprogram, the dummy argument is a dummy procedure. The reference is to that dummy procedure.
2. If Rule 1 does not apply, and the scoping unit contains either an interface body with that name or the name has the **EXTERNAL** attribute, the reference is to an external subprogram.
3. In the scoping unit, if a statement function or internal subprogram has that name, the reference is to that procedure.
4. In the scoping unit, if the name has the **INTRINSIC** attribute, the reference is to the intrinsic procedure with that name.
5. The scoping unit contains a reference to a name that is the name of a module procedure that is accessed through use association. Because of possible renaming in the **USE** statement, the name of the reference may differ from the original procedure name.
6. If none of these rules apply, the reference is resolved by applying these rules to the host scoping unit.

The following rules are used to resolve a procedure reference to a name that is not established:

1. If the scoping unit is a subprogram and if the name is the name of a dummy argument of that subprogram, the dummy argument is a dummy procedure. The reference is to that dummy procedure.
2. If Rule 1 does not apply, and the name is the name of an intrinsic procedure, the reference is to that intrinsic procedure. For this rule to apply, there must be agreement between the intrinsic procedure definition and the reference that the name is either a function or subroutine.
3. If neither Rule 1 nor 2 applies, the reference is to the external procedure with that name.

Resolving Procedure References to Generic Names

When resolving a procedure reference to a generic name, the following rules are followed:

- If the reference is consistent with one of the specific interfaces within a generic interface of the same name, and either appears in the same scoping

unit in which the reference appears or is made accessible by a **USE** statement in the scoping unit, then the reference is to that specific procedure.

- If the first rule fails then, if the reference is consistent with an elemental reference to one of the specific interfaces within a generic interface of the same name, and either appears in same scoping unit in which the reference appears or is made accessible by a **USE** statement in the scoping unit, then the reference is to the specific elemental procedure in that interface block that provides that interface.
- If the previous two rules fail then, if the scoping unit contains for that name either an **INTRINSIC** attribute specification or the name is made accessible from a module in which the corresponding name is specified to have the **INTRINSIC** attribute, and if the interface of that intrinsic procedure is consistent with the reference, the reference will be to that intrinsic procedure.
- If the previous three rules fail then, if the scoping unit has a host scoping unit in which the name is established to be generic within it, and there is an agreement between the units on whether the name is a function or subroutine name, the name will be resolved by applying these rules to the host scoping unit.

Recursion

A procedure that can reference itself, directly or indirectly, is called a recursive procedure. Such a procedure can reference itself indefinitely until a specific condition is met. For example, you can determine the factorial of the positive integer `N` as follows:

```
INTEGER N, RESULT, FACTORIAL
READ (5,*) N
IF (N.GE.0) THEN
  RESULT = FACTORIAL(N)
END IF
CONTAINS
  RECURSIVE FUNCTION FACTORIAL (N) RESULT (RES)
    INTEGER RES
    IF (N.EQ.0) THEN
      RES = 1
    ELSE
      RES = N * FACTORIAL(N-1)
    END IF
  END FUNCTION FACTORIAL
END
```

For details on syntax and rules, see “FUNCTION” on page 320, “SUBROUTINE” on page 396, or “ENTRY” on page 303.

You can also call external procedures recursively when you specify the **-qrecur** compiler option, although XL Fortran disregards this option if the procedure specifies either the **RECURSIVE** or **RESULT** keyword.

Pure Procedures

Because **PURE** procedures are free of side effects, the compiler is not constrained to invoke them in any particular order. Exceptions to this are as follows:
- a pure function, because a value is returned
- a pure subroutine, because you can modify dummy arguments with an intent of **OUT** or **INOUT** or modify the association status or the value of dummy arguments with the **POINTER** attribute

Pure procedures are particularly useful in **FORALL** statements and constructs, which by design require that all referenced procedures be free of side effects.

A procedure must be pure in the following contexts:
- An internal procedure of a pure procedure
- A procedure referenced in the *scalar_mask_expr* or body of a **FORALL** statement or construct, including one referenced by a defined operator or defined assignment
- A procedure referenced in a pure procedure
- A procedure actual argument to a pure procedure

Intrinsic functions (except **RAND**, an XL Fortran extension) and the **MVBITS** subroutine are always pure. They do not need to be explicitly declared to have the **PURE** attribute. A statement function is pure if and only if all functions that it references are pure.

The *specification_part* of a pure function must specify that all dummy arguments have an intent of **IN**, except procedure arguments, and arguments with the **POINTER** attribute. The *specification_part* of a pure subroutine must specify the intents of all dummy arguments, except for procedure arguments, asterisks, and arguments that have the **POINTER** argument. Any interface body for such pure procedures must similarly specify the intents of its dummy arguments.

The *execution_part* and *internal_subprogram_part* of a pure procedure cannot refer to a dummy argument with an intent of **IN**, a global variable (or any object that is storage associated with one), or any subobject thereof, in contexts that may cause its value to change: that is, in contexts that produce side effects. The *execution_part* and *internal_subprogram_part* of a pure function must not use a dummy argument, a global variable, or an object that is storage associated with a global variable, or a subobject thereof, in the following contexts:

- As *variable* in an assignment statement, or as *expression* in an assignment statement if *variable* is of a derived type that has a pointer component at any level
- As *pointer_object* or *target* in a pointer assignment statement
- As a **DO** or implied-**DO** variable
- As an *input_item* in a **READ** statement
- As an internal file identifier in a **WRITE** statement
- As an **IOSTAT=** or **SIZE=** specifier variable in an input/output statement
- As a variable in an **ALLOCATE**, **DEALLOCATE**, **NULLIFY**, or **ASSIGN** statement
- As an actual argument that is associated with a dummy argument with the **POINTER** attribute or with an intent of **OUT** or **INOUT**
- As the argument to **LOC**
- As a **STAT=** specifier
- As a variable in a **NAMELIST** which appears in a **READ** statement

A pure procedure must not specify that any entity is **VOLATILE**. In addition, it must not contain any references to data that is **VOLATILE**, that would otherwise be accessible through use- or host-association. This includes references to data which occur through **NAMELIST I/O**.

Only internal input/output is permitted in pure procedures. Therefore, the unit identifier of an input/output statement cannot be an asterisk (*) or refer to an external unit. The input/output statements are **BACKSPACE**, **ENDFILE**, **REWIND**, **OPEN**, **CLOSE**, **INQUIRE**, **READ**, **PRINT**, and **WRITE**. The **PAUSE** and **STOP** statements are not permitted in pure procedures.

There are two differences between pure functions and pure subroutines:

1. Subroutine nonpointer dummy data objects may have any intent, while function nonpointer dummy data objects must have an intent of **IN**
2. Subroutine dummy data objects with the **POINTER** attribute may change association status and/or definition status

If a procedure is not defined as pure, it must not be declared pure in an interface body. However, the converse is not true: if a procedure is defined as pure, it does not need to be declared pure in an interface body. Of course, if an interface body does not declare that a procedure is pure, that procedure (when referenced through that explicit interface) cannot be used as a reference where only pure procedure references are permitted (for example, in a **FORALL** statement).

Examples

```
PROGRAM ADD
  INTEGER ARRAY(20,256)
  INTERFACE                          ! Interface required for
    PURE FUNCTION PLUS_X(ARRAY)      !   a pure procedure
```

```
      INTEGER, INTENT(IN) :: ARRAY(:)
      INTEGER :: PLUS_X(SIZE(ARRAY))
    END FUNCTION
  END INTERFACE
  INTEGER :: X
  X = ABS(-4)                            ! Intrinsic function
                                         !   is always pure
  FORALL (I=1:20, I /= 10)
    ARRAY(I,:) = I + PLUS_X(ARRAY(I,:))  ! Procedure references in
                                         !   FORALL must be pure
  END FORALL
END PROGRAM
PURE FUNCTION PLUS_X(ARRAY)
  INTEGER, INTENT(IN) :: ARRAY(:)
  INTEGER :: PLUS_X(SIZE(ARRAY)),X
  INTERFACE
    PURE SUBROUTINE PLUS_Y(ARRAY)
      INTEGER, INTENT(INOUT) :: ARRAY(:)
    END SUBROUTINE
  END INTERFACE
  X=8
  PLUS_X = ARRAY+X
  CALL PLUS_Y(PLUS_X)
END FUNCTION

PURE SUBROUTINE PLUS_Y(ARRAY)
  INTEGER, INTENT(INOUT) :: ARRAY(:)     ! Intent must be specified
  INTEGER :: Y
  Y=6
  ARRAY = ARRAY+Y
END SUBROUTINE
```

Elemental Procedures

An elemental subprogram definition must have the **ELEMENTAL** prefix specifier. If the **ELEMENTAL** prefix specifier is used, the **RECURSIVE** specifier cannot be used.

You cannot use the **-qrecur** option when specifying elemental procedures.

An elemental subprogram is a pure subprogram. However, pure subprograms are not necessarily elemental subprograms. For elemental subprograms, it is not necessary to specify both the **ELEMENTAL** prefix specifier and the **PURE** prefix specifier; the **PURE** prefix specifier is implied by the presence of the **ELEMENTAL** prefix specifier. A standard conforming subprogram definition or interface body can have both the **PURE** and **ELEMENTAL** prefix specifiers.

Elemental procedures, subprograms, and user-defined elemental procedures must conform to the following rules:

- The result of an elemental function must be a scalar, and cannot have the **POINTER** attribute.

- The following apply to dummy arguments used in elemental subprograms:
  - All dummy arguments must be scalar, and cannot have the **POINTER** attribute.
  - A dummy argument, or a suboject thereof, cannot be used in a specification expression, except if it is used as an argument to the **BIT_SIZE**, **KIND**, or **LEN** intrinsic functions, or as an argument to one of the numeric inquiry intrinsic functions, see "Chapter 12. Intrinsic Procedures" on page 519.
  - A dummy argument cannot be an asterisk.
  - A dummy argument cannot be a dummy procedure.
- Elemental subprograms must also follow all of the rules that apply to pure subprograms, defined in "Pure Procedures" on page 176.
- Elemental subprograms must follow the rules that apply to pure procedures, described in "Program Units, Procedures, and Subprograms" on page 141.
- Elemental subprograms can have **ENTRY** statements, but the **ENTRY** statement cannot have the **ELEMENTAL** prefix. The procedure defined by the **ENTRY** statement is elemental if the **ELEMENTAL** prefix is specified in the **SUBROUTINE** or **FUNCTION** statement.
- Elemental procedures can be used as defined operators in elemental expressions, but they must follow the rules for elemental expressions as described in "Operators and Expressions" on page 90.

A reference to an elemental procedure is elemental only if:

- The reference is to an elemental function, one or more of the actual arguments is an array, and all array actual arguments have the same shape; or
- The reference is to an elemental subroutine, and all actual arguments that correspond to the **INTENT(OUT)** and **INTENT(INOUT)** dummy arguments are arrays that have the same shape. The remaining actual arguments are conformable with them.

A reference to an elemental subprogram is not elemental if all of its arguments are scalar.

The actual arguments in a reference to an elemental procedure can be either of the following:

- All scalar. For elemental functions, if the arguments are all scalar, the result is scalar.
- One or more array-valued. The following rules apply if one or more of the arguments is array-valued:

- For elemental functions, the shape of the result is the same as the shape of the array actual argument with the greatest rank. If more than one argument appears then all actual arguments must be conformable.
- For elemental subroutines, all actual arguments associated with **INTENT(OUT)** and **INTENT(INOUT)** dummy arguments must be arrays of the same shape, and the remaining actual arguments must be conformable with them.

For elemental references, the resulting values of the elements are the same as would be obtained if the subroutine or function had been applied separately in any order to the corresponding elements of each array actual argument.

If the intrinsic subroutine **MVBITS** is used, the arguments that correspond to the **TO** and **FROM** dummy arguments may be the same variable. Apart from this, the actual arguments in a reference to an elemental subroutine or elemental function must satisfy the restrictions described in "Argument Association" on page 163.

Special rules apply to generic procedures that have an elemental specific procedure, see "Resolving Procedure References to Generic Names" on page 174.

Examples

Example 1:

```
! Example of an elemental function
INTERFACE
  ELEMENTAL REAL FUNCTION LOGN(X,N)
     REAL, INTENT(IN) :: X
     INTEGER, INTENT(IN) :: N
  END FUNCTION LOGN
END INTERFACE

REAL RES(100), VAL(100,100)
  ...
  DO I=1,100
     RES(I) = MAXVAL( LOGN(VAL(I,:),2) )
  END DO
    ...
END PROGRAM P
```

Example 2:

```
! Elemental procedure declared with a generic interface
INTERFACE RAND
   ELEMENTAL FUNCTION SCALAR_RAND(x)
     REAL, INTENT(IN) :: X
   END FUNCTION SCALAR_RAND

   FUNCTION VECTOR_RANDOM(x)
     REAL X(:)
```

```
     REAL VECTOR_RANDOM(SIZE(x))
   END FUNCTION VECTOR_RANDOM
END INTERFACE RAND

REAL A(10,10), AA(10,10)

! Since the actual argument 'AA' is a two-dimensional array, and
! a specific procedure that takes a two-dimensional array as an
! argument is not declared in the interface block, then the
! elemental procedure 'SCALAR_RAND'  is called.

A = RAND(AA)

! Since the actual argument is a one-dimensional array section, and
! a specific procedure that takes a one-dimensional array as an
! argument is declared in the interface block, then the specific
! one-dimensional procedure 'VECTOR_RANDOM' is called. This is an
! non-elemental reference since 'VECTOR_RANDOM' is not declared to
! be elemental.

A = RAND(AA(6:10,2))

END
```

Chapter 8. Input/Output Concepts

This chapter describes:

- "Records"
- "Files" on page 184
- "Units" on page 186
- "Conditions and IOSTAT Values" on page 193

Records

A record is a sequence of characters or a sequence of values. The three kinds of records are formatted, unformatted, and endfile.

Formatted Records

A formatted record is a sequence of any ASCII characters that can be printed in a readable form. When a formatted record is read, data values represented by characters are converted to an internal form. When a formatted record is written, the data to be written is converted from internal form to characters.

If a formatted record is printed using the AIX **asa** command, the first character of the record determines vertical spacing and is not printed. See "Printing Output Files with Fortran ASA Carriage Controls (asa)" in the *XL Fortran for AIX User's Guide* for details. The remaining characters of the record, if any, are printed beginning at the left margin. Vertical spacing can be specified in a format specification in the form of literal data. Vertical spacing is as follows:

First Character of Record	Vertical Spacing Before Printing
Blank	One line
0	Two lines
1	To first line of next page
+	No advance

The characters and spacing shown are those defined for print records. If you use any other character as the first character of the record, an error is produced. If the print record contains no characters, spacing is advanced by one line and a blank line is printed. If records are to be displayed at a terminal, the first character of the record is also used, but only the characters blank, 0, and + produce the spacing shown. (The AIX **asa** command must be used if these print codes are to be displayed on a terminal. See "Printing Output Files with Fortran ASA Carriage Controls (asa)" in the *User's Guide* for details.)

© Copyright IBM Corp. 1990, 2000

Unformatted Records

An unformatted record is a sequence of values in an internal representation that can contain both character and noncharacter data or can contain no data. The values are in their internal form and are not converted in any way when read or written.

Endfile Records

If it exists, an endfile record is the last record of a file. It has no length. It can be written explicitly by an **ENDFILE** statement. It can be written implicitly to a file connected for sequential access when the last data transfer statement was a **WRITE** statement, no intervening file positioning statement referring to the file has been executed, and the following is true:

- A **REWIND** or **BACKSPACE** statement references the unit to which the file is connected; or
- The file is closed, either explicitly by a **CLOSE** statement, implicitly by a program termination not caused by an error condition, or implicitly by another **OPEN** statement for the same unit.

Files

A file is a sequence of records. The two kinds of files are external and internal. Access to an external file can be sequential or direct.

External Files

An external file is associated with an input/output device, such as a disk, tape, or terminal.

An external file is said to exist for an executable program if it is available to the program for reading or writing, or was created within the program. Creating an external file causes it to exist. Deleting an external file ends its existence. An external file can exist without containing any records.

If an external file is identified by a file name, a valid AIX operating system file name must have a full path name of total length ≤1023 characters, with each file name ≤255 characters long (though the full path name need not be specified).

The position of an external file is usually established by the preceding input/output operation. An external file can be positioned to:

- The initial point, which is the position just before the first record.
- The terminal point, which is the position just after the last record.
- The current record, when the file is positioned within a record. Otherwise, there is no current record.
- The preceding record, which is the record just before the current record. If there is no current record, the preceding record is the record just before the

current file position. A preceding record does not exist when the file is positioned at its initial point or within the first record of the file.

- The next record, which is the record just after the current record. If there is no current record, the next record is the record just after the current position. The next record does not exist when the file is positioned at the terminal point or within the last record of the file.
- An indeterminate position after an error.

External File Access Modes: Sequential or Direct

The two methods of accessing the records of an external file are sequential and direct. The method is determined when the file is connected to a unit.

A file connected for sequential access contains records in the order they were written. The records must be either all formatted or all unformatted; the last record of the file must be an endfile record. The records must not be read or written by direct access input/output statements during the time the file is connected for sequential access.

The records of a file connected for direct access can be read or written in any order. The records must be either all formatted or all unformatted; the last record of the file can be an endfile record if the file was previously connected for sequential access. In this case, the endfile record is not considered a part of the file when it is connected for direct access. The records must not be read or written using sequential access, list-directed formatting, namelist formatting, or a nonadvancing input/output statement.

Each record in a file connected for direct access has a record number that identifies its order in the file. The record number is an integer value that must be specified when the record is read or written. Records are numbered sequentially. The first record is number 1. Records need not be read or written in the order of their record numbers. For example, records 9, 5, and 11 can be written in that order without writing the intermediate records.

All records in a file connected for direct access must have the same length, which is specified in the **OPEN** statement when the file is connected (see “OPEN” on page 355).

Records in a file connected for direct access cannot be deleted, but they can be rewritten with a new value. A record cannot be read unless it has first been written.

Internal Files

An internal file is a character variable that is not an array section with a vector subscript. An internal file always exists.

If an internal file is a scalar character variable, the file consists of one record with a length equal to that of the scalar variable. If an internal file is a character array, each element of the array is a record of the file, with each record having the same length.

Reading and writing records are accomplished by sequential access formatted input/output statements. **READ** and **WRITE** are the only input/output statements that can specify an internal file.

If a **WRITE** statement writes less than an entire record, blanks fill the remainder of the record.

On input, blanks are treated in the same way as for an external file opened with **BLANK=NULL** specified. Records are padded with blanks as required, unless the **noblankpad** suboption of the **-qxlf77** compiler option is specified, which indicates that records are not padded.

A scalar character variable that is a record of an internal file can become defined or undefined by means other than an output statement. For example, you can define it by a character assignment statement.

Units

A unit is a means of referring to an external file. Programs refer to external files by the unit numbers indicated by unit specifiers in input/output statements. See page 375 for the form of a unit specifier.

Connection of a Unit

The association of a unit with an external file is called a connection. A connection must occur before the records of the file can be read or written.

There are three ways to connect a file to a unit:
- Preconnection
- Implicit connection
- Explicit connection, using the **OPEN** statement (see page 355)

Preconnection

Preconnection occurs once the program begins executing. Preconnected units can be specified in input/output statements without the prior execution of an **OPEN** statement. Units 0, 5, and 6 are preconnected to unnamed files for formatted sequential access:
- Unit 0 is preconnected to the standard error device
- Unit 5 is preconnected to the standard input device
- Unit 6 is preconnected to the standard output device

The other properties for these files are the default specifier values for **OPEN** specifiers, except the following:

- **STATUS='OLD'**
- **ACTION='READWRITE'**
- **FORM='FORMATTED'**

Implicit Connection

A file with a predetermined name becomes implicitly connected to a unit (by default, unit *n* to a file named **fort**.*n*) when a sequential **PRINT**, **READ**, **WRITE**, **REWIND**, or **ENDFILE** statement is executed on a unit that is not currently connected to an external file. Only unit 0 cannot be implicitly connected. These files need not exist and are created if you use their units without first performing an **OPEN** statement. The default connection is for sequential input/output. (To implicitly connect to a different file name, see the **UNIT_VARS** run-time option under "Setting Runtime Options for Input/Output" in the *User's Guide*.)

A preconnected unit can only be implicitly connected if the connection of the unit to the external file was terminated. In the next example, the preconnected unit is closed before implicit connection can take place:

```
      PROGRAM TRYME
      WRITE ( 6, 10 ) "Hello1"   ! "Hello1" written to standard output
      CLOSE ( 6 )
      WRITE ( 6, 10 ) "Hello2"   ! "Hello2" written to fort.6
10    FORMAT (A)
      END
```

The properties of an implicitly connected unit are the default specifier values for the **OPEN** statement, except for the **FORM=** and **ASYNCH=** specifiers, whose value is determined by the first data transfer statement. The value of the **FORM=** specifier is set to **FORMATTED** when the first input/output statement uses format-directed, list-directed or namelist formatting; and is set to **UNFORMATTED** when the first input/output statement is unformatted. The value of the **ASYNCH=** specifier is set to **YES** when the first input/output statement is asynchronous; and is set to **NO** when the first input/output statement is synchronous.

Disconnection

The **CLOSE** statement disconnects a file from a unit. The file can be connected again within the same program to the same unit or to a different unit, and the unit can be connected again within the same program to the same file or a different file.

Unit 0 cannot be closed. Units 5 and 6 cannot be reconnected to standard input and standard output, respectively, after having been closed.

Executing Data Transfer Statements

The **READ** statement obtains data from an external or internal file and places it in internal storage. Values are transferred from the file to the data items specified by the input list, if one is specified.

The **WRITE** statement places data obtained from internal storage into an external or internal file. The **PRINT** statement places data obtained from internal storage into an external file. Values are transferred to the file from the data items specified by the output list and format specification, if you specify them. Execution of a **WRITE** or **PRINT** statement for a file that does not exist creates the file, unless an error occurs.

If the output list is omitted in a **PRINT** statement, a blank record is transmitted to the output device unless the **FORMAT** statement referred to contains as its first specification a character string edit descriptor or a slash edit descriptor. In this case, the records indicated by these specifications are transmitted to the output device.

Zero-sized arrays and implied-**DO** lists with iteration counts of zero are ignored when determining the next item to be processed. Zero-length scalar character items are not ignored.

If an input/output item is a pointer, data is transferred between the file and the associated target.

During advancing input from a file whose **PAD=** specifier has the value **NO**, the input list and format specification must not require more characters from the record than the record contains. If the **PAD=** specifier has the value **YES** or if the input file is an internal file, blank characters are supplied if the input list and format specification require more characters from the record than the record contains.

If you want only external files connected for sequential access to be padded, specify the **noblankpad** suboption of the **-qxlf77** compiler option, which also sets the default value for the **PAD=** specifier to **NO** for direct files and **YES** for sequential files.

During nonadvancing input from a file whose **PAD=** specifier has the value **NO**, an end-of-record condition occurs if the input list and format specification require more characters from the record than the record contains. If the **PAD=** specifier has the value **YES**, an end-of-record condition occurs and blank characters are supplied if an input item and its corresponding data edit descriptor require more characters from the record than the record contains.

Executing Data Transfer Statements Asynchronously

Synchronous Input/Output (I/O) halts the execution of an application until the I/O operation is complete. Asynchronous I/O allows an application to continue processing while the I/O operation is performed in the background.

Fortran asynchronous **READ** and **WRITE** data transfer statements are used to initiate asynchronous data transfer. Execution continues after the asynchronous I/O statement whether or not the actual data transfer is complete. The execution of the data transfer statement must eventually be followed by the execution of a matching **WAIT** statement specifying the same **ID=** value that was returned to the **ID=** variable in the data transfer statement. See "WAIT" on page 412 for the definition of the matching **WAIT** statement.

The actual data transfer of an I/O item specified in an asynchronous I/O statement may be completed:
- During the asynchronous data transfer statement;
- At any time before the execution of the matching **WAIT** statement; or,
- During the matching **WAIT** statement.

There are, however, situations where the actual data transfer must be completed during the asynchronous data transfer statement. For more information on these types of situations, see "AIX Implementation Details of XL Fortran Input/Output" in the *User's Guide*.

If an error occurs during the data transfer statement, the matching **WAIT** statement will not be required because the **ID=** value will not be defined. Otherwise, error handling and status reporting (**ERR=** and **IOSTAT=**) are performed as if the data transfer statement had been executed synchronously in place of the matching **WAIT** statement.

Any variable that appears as an I/O list item in an asynchronous data transfer statement, or that is associated with such a variable, must not be referenced, become defined, or become undefined until the execution of the matching **WAIT** statement.

Any deallocation of allocatable arrays and pointers and changing association status of pointers are also disallowed between an asynchronous data transfer statement and the matching **WAIT** statement.

Multiple outstanding asynchronous data transfer operations on the same unit are allowed, but they must all be **READ**s or all be **WRITE**s. No other I/O statements on the same unit are allowed until the matching **WAIT** statements for all outstanding asynchronous data transfer operations on the same unit are executed. In the case of direct access, an asynchronous **WRITE** statement must

not specify both the same unit and record number as any asynchronous **WRITE** statement for which the matching **WAIT** statement has not been executed.

In the portion of the program that executes between the asynchronous data transfer statement and the matching **WAIT** statement, the *integer_variable* in the **NUM=** specifier or any variable associated with it must not be referenced, become defined, or become undefined.

Example:

```
! A program demonstrating the use of asynchronous I/O statements.

SUBROUTINE COMPARE(ISTART, IEND, ISIZE, A)
INTEGER, DIMENSION(ISIZE) :: A
INTEGER I, ISTART, IEND, ISIZE
DO I = ISTART, IEND
  IF (A (I) /= I) THEN
    PRINT *, "Expected ", I, ", got ", A(I)
  END IF
END DO
END SUBROUTINE COMPARE

PROGRAM SAMPLE
INTEGER, PARAMETER :: ISIZE = 1000000
INTEGER, PARAMETER :: SECT1 = (ISIZE/2) - 1, SECT2 = ISIZE - 1
INTEGER, DIMENSION(ISIZE), STATIC :: A
INTEGER IDVAR

OPEN(10, STATUS="OLD", ACCESS="DIRECT", ASYNCH="YES", RECL=(ISIZE/2)*4)
A = 0

! Reads in the first part of the array.

READ(10, REC=1) A(1:SECT1)

! Starts asynchronous read of the second part of the array.

READ(10,ID=IDVAR, REC=2) A(SECT1+1:SECT2)

! While the second asynchronous read is being performed,
! do some processing here.

CALL COMPARE(1, SECT1, ISIZE, A)

WAIT(ID=IDVAR)

CALL COMPARE(SECT1+1, SECT2, ISIZE, A)
END
```

Related Information:

- "AIX Implementation Details of XL Fortran Input/Output" in the *User's Guide*

- “READ” on page 374
- “WAIT” on page 412
- “WRITE” on page 416

Advancing and Nonadvancing Input/Output

Advancing input/output positions the file after the last record that is read or written, unless an error condition is encountered.

Nonadvancing input/output can position the file at a character position within the current record, or a subsequent record. With nonadvancing input/output, you can read or write a record of the file by a sequence of input/output statements that each access a portion of the record. You can also read variable-length records and inquire about their lengths.

```
! Reads digits using nonadvancing input

    INTEGER COUNT
    CHARACTER(1) DIGIT
    OPEN (7)
    DO
     READ (7,FMT="(A1)",ADVANCE="NO",EOR=100) DIGIT
       COUNT = COUNT + 1
     IF ((ICHAR(DIGIT).LT.ICHAR('0')).OR.(ICHAR(DIGIT).GT.ICHAR('9'))) THEN
       PRINT *,"Invalid character ", DIGIT, " at record position ",COUNT
       STOP
     END IF
    END DO
100 PRINT *,"Number of digits in record = ", COUNT
    END
```

File Position Before and After Data Transfer

For an explicit connection (by an **OPEN** statement) for sequential input/output that specifies the **POSITION=** specifier, the file position can be explicitly positioned at the beginning, at the end, or where the position is on opening.

If the **OPEN** statement does not specify the **POSITION=** specifier:
- If the **STATUS=** specifier has the value **NEW** or **SCRATCH**, the file is positioned at the beginning.
- If **STATUS='OLD'** is specified, the **-qposition=appendold** compiler option is specified, and the next operation that changes the file position is a **WRITE** statement, then the file is positioned at the end. If all these conditions are not met, the file is positioned at the beginning.
- If **STATUS='UNKNOWN'** is specified, the **-qposition=appendunknown** compiler option is specified, and the next operation is a **WRITE** statement, then the file is positioned at the end. If all these conditions are not met, the file is positioned at the beginning.

After an implicit **OPEN**, the file is positioned at the beginning. Thus:

- If the first input/output operation on the file is a **READ**, it will read the first record of the file.
- If the first input/output operation on the file is a **WRITE**, it will delete the contents of the file and write at the first record.

A **REWIND** statement can be used to position a file at its beginning. The preconnected units 0, 5 and 6 are positioned as they come from the program's parent process.

The positioning of a file prior to data transfer depends on the method of access:

- Sequential access for an external file:
  - For advancing input, the file is positioned at the beginning of the next record. This record becomes the current record.
  - For advancing output, a new record is created and becomes the last record of the file.
- Sequential access for an internal file: the file is positioned at the beginning of the first record of the file. This record becomes the current record.
- Direct access: the file is positioned at the beginning of the record specified by the record specifier. This record becomes the current record.

After advancing input/output data transfer, the file is positioned:

- Beyond the endfile record if an end-of-file condition exists as a result of reading an endfile record.
- Beyond the last record read or written if no error or end-of-file condition exists. That last record becomes the preceding record. A record written on a file connected for sequential access becomes the last record of the file.

For nonadvancing input, if no error condition or end-of-file condition occurs, but an end-of-record condition occurs, the file is positioned just after the record read. If no error condition, end-of-file condition or end-of-record condition occurs in a nonadvancing input statement, the file position does not change. If no error condition occurs in a nonadvancing output statement, the file position is not changed. In all other cases, the file is positioned just after the record read or written and that record becomes the preceding record.

If a file is positioned beyond the endfile record, a **READ**, **WRITE**, **PRINT**, or **ENDFILE** statement cannot be executed (assuming **-qxlf77=softeof** is not set). A **BACKSPACE** or **REWIND** statement can be used to reposition the file.

Use the **-qxlf77=softeof** option to be able to read and write past the end-of-file. See "-qxlf77 Option" in the *User's Guide* for details.

Conditions and IOSTAT Values

An IOSTAT value is a value assigned to the variable for the **IOSTAT=** specifier if an end-of-file condition, end-of-record condition or an error condition occurs during an input/output statement. There are five types of error conditions: catastrophic, severe, recoverable, conversion, Fortran 90 and Fortran 95 language.

End-Of-Record Conditions

An end-of-record condition causes the **IOSTAT=** specifier to be set to -4 and the **EOR=** label to be branched to if these specifiers are present on the input/output statement. If the **IOSTAT=** and **EOR=** specifiers are not present on the input/output statement when an end-of-record condition is encountered, the program stops.

Table 6. IOSTAT Values for End-Of-Record Conditions

IOSTAT Value	End-of-Record Condition Description
-4	End of record encountered on a nonadvancing, format-directed READ of an external file.

End-Of-File Conditions

An end-of-file condition can occur at the beginning of execution of an input statement or during execution of a formatted input statement when more than one record is required by the interaction of the input list and the format. An end-of-file condition causes the **IOSTAT=** specifier to be set to one of the values defined below and the **END=** label to be branched to if these specifiers are present on the input statement. If the **IOSTAT=** and **END=** specifiers are not present on the input statement when an end-of-file condition is encountered, the program stops.

Table 7. IOSTAT Values for End-Of-File Conditions

IOSTAT Value	End-of-File Condition Description
-1	End of file encountered on sequential READ of an external file, or END= is specified on a direct access read and the record is nonexistent.
-2	End of file encountered on READ of an internal file.

Error Conditions

Catastrophic Errors

Catastrophic errors are system-level errors encountered within the run-time system that prevent further execution of the program. When a catastrophic error occurs, a short (non-translated) message is written to unit 0, followed by a call to the C library routine **abort()**. A core dump may result, depending on how your execution environment is configured.

Severe Errors

A severe error cannot be recovered from, even if the **ERR_RECOVERY** run-time option has been specified with the value **YES**. A severe error causes the **IOSTAT=** specifier to be set to one of the values defined below and the **ERR=** label to be branched to if these specifiers are present on the input/output statement. If the **IOSTAT=** and **ERR=** specifiers are not present on the input/output statement when a severe error condition is encountered, the program stops.

Table 8. IOSTAT Values for Severe Error Conditions

IOSTAT Value	Error Description
1	END= is not specified on a direct access READ and the record is nonexistent.
2	End of file encountered on WRITE of an internal file.
6	File cannot be found and STATUS='OLD' is specified on an OPEN statement.
10	Read error on direct file.
11	Write error on direct file.
12	Read error on sequential file.
13	Write error on sequential file.
14	Error opening file.
15	Permanent I/O error encountered on file.
37	Dynamic memory allocation failure - out of memory.
38	REWIND error.
39	ENDFILE error.
40	BACKSPACE error.
107	File exists and STATUS='NEW' was specified on an OPEN statement.
119	BACKSPACE statement attempted on unit connected to a tape device.
122	Incomplete record encountered during direct access READ.
130	ACTION='READWRITE' specified on an OPEN statement to connect a pipe.
135	The user program is making calls to an unsupported version of the XL Fortran Run-time Environment.
139	I/O operation not permitted on the unit because the file was not opened with an appropriate value for the ACTION= specifier.
142	CLOSE error.
144	INQUIRE error.

Table 8. IOSTAT Values for Severe Error Conditions (continued)

IOSTAT Value	Error Description
152	ACCESS='DIRECT' is specified on an OPEN statement for an AIX file that can only be accessed sequentially.
153	POSITION='REWIND' or POSITION='APPEND' is specified on an OPEN statement and the file is an AIX pipe.
156	Invalid value for RECL= specifier on an OPEN statement.
159	External file input could not be flushed because the associated device is not seekable.
165	The record number of the next record that can be read or written is out of the range of the variable specified with the NEXTREC= specifier of the INQUIRE statement.
169	The asynchronous I/O statement cannot be completed because the unit is connected for synchronous I/O only.
172	The connection failed because the file does not allow asynchronous I/O.
173	An asynchronous READ statement was executed while asynchronous WRITE statements were pending for the same unit, or an asynchronous WRITE statement was executed while asynchronous READ statements were pending for the same unit.
174	The synchronous I/O statement cannot be completed because an earlier asynchronous I/O statement has not been completed.
175	The WAIT statement cannot be completed because the value of the ID= specifier is invalid.
176	The WAIT statement cannot be completed because the corresponding asynchronous I/O statement is in a different scoping unit.
178	The asynchronous direct WRITE statement for a record is not permitted because an earlier asynchronous direct WRITE statement for the same record has not been completed.
179	The I/O operation cannot be performed on the unit because there are still incomplete asynchronous I/O operations on the unit.
181	A file cannot be connected to a unit because multiple connections are allowed for synchronous I/O only.
182	Invalid value for UWIDTH= option. It must be set to either 32 or 64.
183	The maximum record length for the unit is out of the range of the scalar variable specified with the RECL= specifier in the INQUIRE statement.
184	The number of bytes of data transmitted is out of the range of the scalar variable specified with the SIZE= or NUM= specifier in the I/O statement.

Table 8. IOSTAT Values for Severe Error Conditions (continued)

IOSTAT Value	Error Description
185	A file cannot be connected to two units with different UWIDTH values.
186	Unit numbers must be between 0 and 2,147,483,647.

Recoverable Errors

A recoverable error is an error that can be recovered from. A recoverable error causes the **IOSTAT=** specifier to be set to one of the values defined below and the **ERR=** label to be branched to if these specifiers are present on the input/output statement. If the **IOSTAT=** and **ERR=** specifiers are not present on the input/output statement and the **ERR_RECOVERY** run-time option is set to **YES**, recovery action occurs and the program continues. If the **IOSTAT=** and **ERR=** specifiers are not present on the input/output statement and the **ERR_RECOVERY** option is set to **NO**, the program stops.

Table 9. IOSTAT Values for Recoverable Error Conditions

IOSTAT Value	Error Description
16	Value of REC= specifier invalid on direct I/O.
17	I/O statement not allowed on direct file.
18	Direct I/O statement on an unconnected unit.
19	Unformatted I/O attempted on formatted file.
20	Formatted I/O attempted on unformatted file.
21	Sequential I/O attempted on direct file.
22	Direct I/O attempted on sequential file.
23	Attempt to connect a file that is already connected to another unit.
24	OPEN specifiers do not match the connected file’s attributes.
25	RECL= specifier omitted on an OPEN statement for a direct file.
26	RECL= specifier on an OPEN statement is negative.
27	ACCESS= specifier on an OPEN statement is invalid.
28	FORM= specifier on an OPEN statement is invalid.
29	STATUS= specifier on an OPEN statement is invalid.
30	BLANK= specifier on an OPEN statement is invalid.
31	FILE= specifier on an OPEN or INQUIRE statement is invalid.
32	STATUS=’SCRATCH’ and FILE= specifier specified on same OPEN statement.

Table 9. IOSTAT Values for Recoverable Error Conditions (continued)

IOSTAT Value	Error Description
33	STATUS='KEEP' specified on CLOSE statement when file was opened with STATUS='SCRATCH'.
34	Value of STATUS= specifier on CLOSE statement is invalid.
36	Invalid unit number specified in an I/O statement.
47	A namelist input item was specified with one or more components of nonzero rank.
48	A namelist input item specified a zero-sized array.
58	Format specification error.
93	I/O statement not allowed on error unit (unit 0).
110	Illegal edit descriptor used with a data item in formatted I/O.
120	The NLWIDTH setting exceeds the length of a record.
125	BLANK= specifier given on an OPEN statement for an unformatted file.
127	POSITION= specifier given on an OPEN statement for a direct file.
128	POSITION= specifier value on an OPEN statement is invalid.
129	ACTION= specifier value on an OPEN statement is invalid.
131	DELIM= specifier given on an OPEN statement for an unformatted file.
132	DELIM= specifier value on an OPEN statement is invalid.
133	PAD= specifier given on an OPEN statement for an unformatted file.
134	PAD= specifier value on an OPEN statement is invalid.
136	ADVANCE= specifier value on a READ statement is invalid.
137	ADVANCE='NO' is not specified when SIZE= is specified on a READ statement.
138	ADVANCE='NO' is not specified when EOR= is specified on a READ statement.
145	READ or WRITE attempted when file is positioned after the endfile record.
163	Multiple connections to a file located on a non-random access device are not allowed.
164	Multiple connections with ACTION='WRITE' or ACTION='READWRITE' are not allowed.
170	ASYNCH= specifier value on an OPEN statement is invalid.
171	ASYNCH= specifier given on an OPEN statement is invalid because the FORM= specifier is set to FORMATTED.
177	The unit was closed while there were still incomplete asynchronous I/O operations.

Conversion Errors

A conversion error occurs as a result of invalid data or the incorrect length of data in a data transfer statement. A conversion error causes the **IOSTAT=** specifier to be set to one of the values defined below and the **ERR=** label to be branched to if these specifiers are present on the input/output statement and the **CNVERR** option is set to **YES**. If the **IOSTAT=** and **ERR=** specifiers are not present on the input/output statement, both the **CNVERR** option and the **ERR_RECOVERY** option are set to **YES**, recovery action is performed and the program continues. If the **IOSTAT=** and **ERR=** specifiers are not present on the input/output statement, the **CNVERR** option is set to **YES**, the **ERR_RECOVERY** option is set to **NO**, and the program stops. If **CNVERR** is set to **NO**, the **ERR=** label is never branched to but the **IOSTAT=** specifier may be set, as indicated below.

Table 10. IOSTAT Values for Conversion Error Conditions

IOSTAT Value	Error Description	IOSTAT set if CNVERR=NO
3	End of record encountered on an unformatted file.	no
4	End of record encountered on a formatted external file using advancing I/O.	no
5	End of record encountered on an internal file.	no
7	Incorrect format of list-directed input found in an external file.	yes
8	Incorrect format of list-directed input found in an internal file.	yes
9	List-directed or NAMELIST data item too long for the internal file.	yes
41	Valid logical input not found in external file.	no
42	Valid logical input not found in internal file.	no
43	Complex value expected using list-directed or NAMELIST input in external file but not found.	no
44	Complex value expected using list-directed or NAMELIST input in internal file but not found.	no
45	NAMELIST item name specified with unknown or invalid derived-type component name in NAMELIST input.	no
46	NAMELIST item name specified with an invalid substring range in NAMELIST input.	no
49	List-directed or namelist input contained an invalid delimited character string.	no

Table 10. IOSTAT Values for Conversion Error Conditions (continued)

IOSTAT Value	Error Description	IOSTAT set if CNVERR=NO
56	Invalid digit found in input for B, O or Z format edit descriptors.	no
84	NAMELIST group header not found in external file.	yes
85	NAMELIST group header not found in internal file.	yes
86	Invalid NAMELIST input value found in external file.	no
87	Invalid NAMELIST input value found in internal file.	no
88	Invalid name found in NAMELIST input.	no
90	Invalid character in NAMELIST group or item name in input.	no
91	Invalid NAMELIST input syntax.	no
92	Invalid subscript list for NAMELIST item in input.	no
94	Invalid repeat specifier for list-directed or NAMELIST input in external file.	no
95	Invalid repeat specifier for list-directed or NAMELIST input in internal file.	no
96	Integer overflow in input.	no
97	Invalid decimal digit found in input.	no
98	Input too long for B, O or Z format edit descriptors.	no
121	Output length of NAMELIST item name or NAMELIST group name is longer than the maximum record length or the output width specified by the NLWIDTH option.	yes

Fortran 90 and Fortran 95 Language Errors

A Fortran 90 language error results from the use of XL Fortran extensions to the Fortran 90 language that cannot be detected at compile time. A Fortran 90 language error is considered a severe error when the **LANGLVL** run-time option has been specified with the value **90STD** and the **ERR_RECOVERY** run-time option has either not been set or is set to **NO**. If both **LANGLVL=90STD** and **ERR_RECOVERY=YES** have been specified, the error is considered a recoverable error. If **LANGLVL= EXTENDED** is specified, the error condition is not considered an error.

A Fortran 95 language error results from the use of XL Fortran extensions to the Fortran 95 language that cannot be detected at compile time. A Fortran 95 language error is considered a severe error when the **LANGLVL** run-time option has been specified with the value **95STD** and the **ERR_RECOVERY**

run-time option has either not been set or is set to **NO**. If both **LANGLVL=95STD** and **ERR_RECOVERY=YES** have been specified, the error is considered a recoverable error. If **LANGLVL=EXTENDED** is specified, the error condition is not considered an error.

Table 11. IOSTAT Values for Fortran 90 and Fortran 95 Language Error Conditions

IOSTAT Value	Error Description
53	Mismatched edit descriptor and item type in formatted I/O.
58	Format specification error.
140	Unit is not connected when the I/O statement is attempted. Only for READ, WRITE, PRINT, REWIND, and ENDFILE.
141	Two ENDFILE statements without an intervening REWIND or BACKSPACE on the unit.
151	The FILE= specifier is missing and the STATUS= specifier does not have a value of 'SCRATCH' on an OPEN statement.
187	NAMELIST comments are not allowed by the Fortran 90 standard.

Chapter 9. Input/Output Formatting

Formatted **READ**, **WRITE**, and **PRINT** statements use formatting information to direct the editing (conversion) between internal data representations and character representations in formatted records (see "FORMAT" on page 315).

This chapter describes:
- "Format-Directed Formatting"
- "Editing" on page 204
- "Interaction between Input/Output Lists and Format Specifications" on page 226
- "List-Directed Formatting" on page 227
- "Namelist Formatting" on page 231

Format-Directed Formatting

In format-directed formatting, editing is controlled by edit descriptors in a format specification. A format specification is specified in a **FORMAT** statement or as the value of a character array or character expression in a data transfer statement.

Data Edit Descriptors

Forms	Use	Page
A **A***w*	Edits character values	205
B*w* **B***w.m*	Edits binary values	206
E*w.d* **E***w.d***E***e* **E***w.d***D***e* **E***w.d***Q***e* **D***w.d* **EN***w.d* **EN***w.d***E***e* **ES***w.d* **ES***w.d***E***e* **Q***w.d*	Edits real and complex numbers with exponents	207
F*w.d*	Edits real and complex numbers without exponents	211
G*w.d* **G***w.d***E***e* **G***w.d***D***e* **G***w.d***Q***e*	Edits data fields of any intrinsic type, with the output format adapting to the type of the data and, if the data is of type real, the magnitude of the data	212

© Copyright IBM Corp. 1990, 2000

Forms	Use	Page
I*w* **I***w.m*	Edits integer numbers	214
L*w*	Edits logical values	215
O*w* **O***w.m*	Edits octal values	216
Q	Returns the count of characters remaining in an input record	217
Z*w* **Z***w.m*	Edits hexadecimal values	219

where:

w specifies the width of a field, including all blanks. It must be positive, except that it can be zero for **I**, **B**, **O**, **Z**, and **F** edit descriptors on output.

m specifies the number of digits to be printed

d specifies the number of digits to the right of the decimal point

e specifies the number of digits in the exponent field

w, *m*, *d*, and *e* can be:
- An unsigned integer literal constant
- A scalar integer expression enclosed by angle brackets (< and >). See "Variable Format Expressions" on page 320 for details.

You cannot specify kind parameters for *w*, *m*, *d*, or *e*.

Note:

There are two types of **Q** data edit descriptor (**Q***w.d* and **Q**):

extended precision Q
is the **Q** edit descriptor whose syntax is **Q***w.d*

character count Q
is the **Q** edit descriptor whose syntax is **Q**

Control Edit Descriptors

Forms	Use	Page
/ *r* /	Specifies the end of data transfer on the current record	220
:	Specifies the end of format control if there are no more items in the input/output list	221

Forms	Use	Page
$	Suppresses end-of-record in output	221
BN	Ignores nonleading blanks in numeric input fields	222
BZ	Interprets nonleading blanks in numeric input fields as zeros	222
*k***P**	Specifies a scale factor for real and complex items	224
S SS	Specifies that plus signs are not to be written	224
SP	Specifies that plus signs are to be written	224
T*c*	Specifies the absolute position in a record from which, or to which, the next character is transferred	225
TL*c*	Specifies the relative position (backward from the current position in a record) from which, or to which, the next character is transferred	225
TR*c*	Specifies the relative position (forward from the current position in a record) from which, or to which, the next character is transferred	225
*o***X**	Specifies the relative position (forward from the current position in a record) from which, or to which, the next character is transferred	225

where:

r is a repeat specifier. It is an unsigned, positive, integer literal constant.

k specifies the scale factor to be used. It is an optionally signed, integer literal constant.

c specifies the character position in a record. It is an unsigned, nonzero, integer literal constant.

o is the relative character position in a record. It is an unsigned, nonzero, integer literal constant.

r, *k*, *c*, and *o* can also be expressed as an arithmetic expression enclosed by angle brackets (< and >) that evaluates into an integer value.

Kind type parameters cannot be specified for *r*, *k*, *c*, or *o*.

Character String Edit Descriptors

Forms	Use	Page
*n***H***str*	Outputs a character string (str)	223
'str' *"str"*	Outputs a character string (*str*)	221

n is the number of characters in a literal field. It is an unsigned,

positive, integer literal constant. Blanks are included in character count. A kind type parameter cannot be specified.

Editing

Editing is performed on fields. A field is the part of a record that is read on input or written on output when format control processes one of the data or character string edit descriptors. The field width is the size of the field in characters.

The **I**, **F**, **E**, **EN**, **ES**, **B**, **O**, **Z**, **D**, **G**, and extended precision **Q** edit descriptors are collectively called numeric edit descriptors. They are used to format integer, real, and complex data. The following general rules apply to these edit descriptors:

- On input:
  - Leading blanks are not significant. The interpretation of other blanks is controlled by the **BLANK=** specifier in the **OPEN** statement and the **BN** and **BZ** edit descriptors. A field of all blanks is considered to be zero. Plus signs are optional, although they cannot be specified for the **B**, **O**, and **Z** edit descriptors.
  - In **F**, **E**, **EN**, **ES**, **D**, **G**, and extended precision **Q** editing, a decimal point appearing in the input field overrides the portion of an edit descriptor that specifies the decimal point location. The field can have more digits than can be represented internally.
- On output:
  - Characters are right-justified inside the field. Leading blanks are supplied if the editing process produces fewer characters than the field width. If the number of characters is greater than the field width, or if an exponent exceeds its specified length, the entire field is filled with asterisks.
  - A negative value is prefixed with a minus sign. By default, a positive or zero value is unsigned; it can be prefixed with a plus sign, as controlled by the **S**, **SP**, and **SS** edit descriptors.
  - Depending on whether you specify the **signedzero** or **nosignedzero** suboptions for the **-qxlf90** compiler option the following will result for the **E**, **D**, **Q(Extended Precision)**, **F**, **EN**, **ES** or **G(General Editing)** edit descriptors:
    - when the **signedzero** suboption is chosen, and the internal value is negative or a negative zero on output, a minus sign always be written out to the output field, even if the output value is zero. The Fortran 95 standard requires this behavior.

      Note that in XL Fortran, a **REAL(16)** internal value of zero is never treated as a negative zero.

- when the **nosignedzero** suboption is chosen, and the output value is zero, no minus sign will be written out to the output field, even if the internal value was negative. The Fortran 90 standard requires this behavior, and is consistent with the behavior of XL Fortran Version 5.1.1.

  Note: The **ES** and **EN** edit descriptors will behave the same for both the **signedzero** and **nosignedzero** suboptions when the internal value is non-zero. That is, the minus sign will be printed out whenever the value is negative.

– In XL Fortran, a `NaN` (not a number) is indicated by "`NaNQ`", "`+NaNQ`", "`-NaNQ`", "`NaNS`", "`+NaNS`", or "`-NaNS`". Infinity is indicated by "`INF`", "`+INF`", or "`-INF`".

Note: In the examples of edit descriptors, a lowercase `b` in the `Output` column indicates that a blank appears at that position.

Complex Editing

A complex value is a pair of separate real components. Therefore, complex editing is specified by a pair of edit descriptors. The first one edits the real part of the number, and the second one edits the imaginary part of the number. The two edit descriptors can be the same or different. One or more control edit descriptors can be placed between them, but not data edit descriptors.

Data Edit Descriptors

A (Character) Editing

Forms:

A
A*w*

The **A** edit descriptor directs the editing of character values. It can correspond to an input/output list item of type character or any other type. The kind type parameter of all characters transferred and converted is implied by the corresponding list item.

On input, if *w* is greater than or equal to the length (call it *len*) of the input list item, the rightmost *len* characters are taken from the input field. If the specified field width is less than *len*, the *w* characters are left-justified, with (*len* - *w*) trailing blanks added.

On output, if *w* is greater than *len*, the output field consists of (*w* - *len*) blanks followed by the *len* characters from the internal representation. If *w* is less than or equal to *len*, the output field consists of the leftmost *w* characters from the internal representation.

If *w* is not specified, the width of the character field is the length of the corresponding input/output list item.

B (Binary) Editing

Forms:

B*w*
B*w***.***m*

The **B** edit descriptor directs editing between values of any type in internal form and their binary representation. (A binary digit is either 0 or 1.)

On input, *w* binary digits are edited and form the internal representation for the value of the input list item. The binary digits in the input field correspond to the rightmost binary digits of the internal representation of the value assigned to the input list item. *m* has no effect on input.

On input, *w* must be greater than zero.

On output, *w* can be zero. If *w* is zero, the output field consists of the least number of characters required to represent the output value.

The output field for **B***w* consists of zero or more leading blanks followed by the internal value in a form identical to the binary digits without leading zeros. Note that a binary constant always consists of at least one digit.

The output field for **B***w.m* is the same as for **B***w*, except that the digit string consists of at least *m* digits. If necessary, the digit string is padded with leading zeros. The value of *m* must not exceed the value of *w* unless *w* is zero. If *m* is zero and the value of the internal data is zero, the output field consists of only blank characters, regardless of the sign control in effect.

If *m* is zero, *w* is positive and the value of the internal datum is zero, the output field consists of *w* blank characters. If both *w* and *m* are zero, and the value of the internal datum is zero, the output field consists of only one blank character.

If the **nooldboz** suboption of the **-qxlf77** compiler option is specified (the default), asterisks are printed when the output field width is not sufficient to contain the entire output. On input, the **BN** and **BZ** edit descriptors affect the **B** edit descriptor.

If the **oldboz** suboption of the **-qxlf77** compiler option is specified, the following occurs on output:

- **B***w* is treated as **B***w.m*, with *m* assuming the value that is the minimum of *w* and the number of digits required to represent the maximum possible value of the data item.

- The output consists of blanks followed by at least *m* digits. These are the rightmost digits of the number, zero-filled if necessary, until there are *m* digits. If the number is too large to fit into the output field, only the rightmost *m* digits are output.

If *w* is zero, the **oldboz** suboption will be ignored.

With the **oldboz** suboption, the **BN** and **BZ** edit descriptors do not affect the **B** edit descriptor.

Examples of B Editing on Input

```
Input      Format     Value
111        B3         7
110        B3         6
```

Examples of B Editing on Output

```
Value      Format         Output                  Output
                  (with -qxlf77=oldboz)  (with -qxlf77=nooldboz)
 7          B3            111                     111
 6          B5            00110                   bb110
17          B6.5          b10001                  b10001
17          B4.2          0001                    ****
22          B6.5          b10110                  b10110
22          B4.2          0110                    ****
 0          B5.0          bbbbb                   bbbbb
 2          B0            10                      10
```

E, D, and Q (Extended Precision) Editing

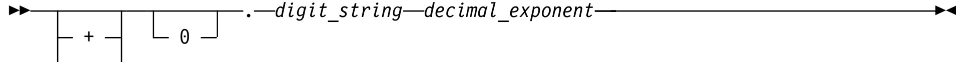
Forms:

E*w*.*d*
E*w*.*d* **E***e*
E*w*.*d* **D***e*
E*w*.*d* **Q***e*
D*w*.*d*
Q*w*.*d*

The **E**, **D**, and extended precision **Q** edit descriptors direct editing between real and complex numbers in internal form and their character representations with exponents. An **E**, **D**, or extended precision **Q** edit descriptor can correspond to an input/output list item of type real, to either part (real or imaginary) of an input/output list item of type complex, or to any other type, as long as the length is at least 4 bytes.

The form of the input field is the same as for **F** editing. *e* has no effect on input.

The form of the output field for a scale factor of 0 is:



digit_string
: is a digit string whose length is the *d* most significant digits of the value after rounding.

decimal_exponent
: is a decimal exponent of one of the following forms (*z* is a digit):

Edit Descriptor	Absolute Value of Exponent (with scale factor of 0)	Form of Exponent
E*w.d*	|decimal_exponent| ≤ 99	**E**$\pm z_1 z_2$
E*w.d*	99<|decimal_exponent| ≤ 309	$\pm z_1 z_2 z_3$
E*w.d***E***e*	|decimal_exponent| ≤ (10^e)-1	**E**$\pm z_1 z_2 \cdots z_e$
E*w.d***D***e*	|decimal_exponent| ≤ (10^e)-1	**D**$\pm z_1 z_2 \cdots z_e$
E*w.d***Q***e*	|decimal_exponent| ≤ (10^e)-1	**Q**$\pm z_1 z_2 \cdots z_e$
D*w.d*	|decimal_exponent| ≤ 99	**D**$\pm z_1 z_2$
D*w.d*	99<|decimal_exponent| ≤ 309	$\pm z_1 z_2 z_3$
Q*w.d*	|decimal_exponent| ≤ 99	**Q**$\pm z_1 z_2$
Q*w.d*	99<|decimal_exponent| ≤ 309	$\pm z_1 z_2 z_3$

The scale factor *k* (see “P (Scale Factor) Editing” on page 224) controls decimal normalization. If $-d<k\leq0$, the output field contains $|k|$ leading zeros and $d - |k|$ significant digits after the decimal point. If $0<k<d+2$, the output field contains *k* significant digits to the left of the decimal point and $d-k+1$ significant digits to the right of the decimal point. You cannot use other values of *k*.

See the general information about numeric editing on page 204 for additional information.

Note: If the value to be displayed using the real edit descriptor is outside of the range of representable numbers, XL Fortran supports the ANSI/IEEE floating-point format by displaying the following:

Table 12. Floating-Point Display

Display	Meaning
NaNQ +NaNQ	Positive Quiet NaN (not-a-number)
-NaNQ	Negative Quiet NaN

Table 12. Floating-Point Display (continued)

Display	Meaning
NaNS +NaNS	Positive Signaling NaN
-NaNS	Negative Signaling NaN
INF +INF	Positive Infinity
-INF	Negative Infinity

Examples of E, D, and Extended Precision Q Editing on Input

(Assume **BN** editing is in effect for blank interpretation.)

```
Input      Format     Value
12.34      E8.4       12.34
·1234E2    E8.4       12.34
2.E10      E12.6E1    2.E10
```

Examples of E, D, and Extended Precision Q Editing on Output

```
Value      Format          Output                     Output
                   (with -qxlf77=noleadzero)  (with -qxlf77=leadzero)
1234.56    E10.3         bb.123E+04                b0.123E+04
1234.56    D10.3         bb.123D+04                b0.123D+04

                   (with -qxlf90=signedzero)  (with -qxlf90=nosignedzero)
-0.001     E5.2          -0.00                     b0.00
```

EN Editing

Forms:
EN*w.d*
EN*w.d***E***e*

The **EN** edit descriptor produces an output field in the form of a real number in engineering notation such that the decimal exponent is divisible by 3 and the absolute value of the significand is greater than or equal to 1 and less than 1000, except when the output value is zero. The scale factor has no effect on output.

The **EN** edit descriptor can correspond to an input/output list item of type real, to either part (real or imaginary) of an input/output list item of type complex, or to any other type, so long as the length is at least 4 bytes.

The form and interpretation of the input field is the same as for F editing.

The form of the output field is:

```
>>─┬───┬─┬───────┬─.─digit_string─exp─────────────────────────><
   ├ + ┤ └ yyy ──┘
   └ - ┘
```

yyy are the 1 to 3 decimal digits representative of the most significant digits of the value of the datum after rounding (*yyy* is an integer such that 1 ≤ *yyy* < 1000 or, if the output value is zero, *yyy* = 0).

digit_string
are the *d* next most significant digits of the value of the datum after rounding.

exp is a decimal exponent, divisible by 3, of one of the following forms (z is a digit):

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
EN*w.d*	|*exp*| ≤ 99	E± $z_1 z_2$
EN*w.d*	99 < |*exp*| ≤ 309	± $z_1 z_2 z_3$
EN*w.d*E*e*	|*exp*| ≤ 10^e–1	E± z_1 ... z_e

For additional information on numeric editing, see "Editing" on page 204.

Examples of EN Editing

```
Value          Format       Output
3.14159        EN12.5       b3.14159E+00
1.41425D+5     EN15.5E4     141.42500E+0003
3.14159D-12    EN15.5E1     ***************

                        (with -qxlf90=signedzero)  (with -qxlf90=nosignedzero)
-0.001         EN9.2        -1.00E-03                   -1.00E-03
```

ES Editing

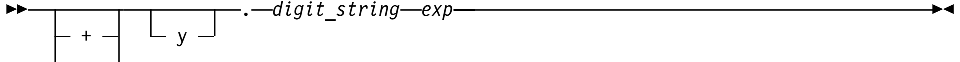
Forms:
ES*w.d*
ES*w.d***E***e*

The **ES** edit descriptor produces an output field in the form of a real number in scientific notation such that the absolute value of the significand is greater than or equal to 1 and less than 10, except when the output value is zero. The scale factor has no effect on output.

The **ES** edit descriptor can correspond to an input/output list item of type real, to either part (real or imaginary) of an input/output list item of type complex, or to any other type, so long as the length is at least 4 bytes.

The form and interpretation of the input field is the same as for F editing.

The form of the output field is:



y is a decimal digit representative of the most significant digit of the value of the datum after rounding.

digit_string
are the *d* next most significant digits of the value of the datum after rounding.

exp is a decimal exponent having one of the following forms (z is a digit):

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
ES*w.d*	$\|exp\| \leq 99$	$E\pm z_1 z_2$
ES*w.d*	$99 < \|exp\| \leq 309$	$\pm z_1 z_2 z_3$
ES*w.d*E*e*	$\|exp\| \leq 10^e-1$	$E\pm z_1 \dots z_e$

For additional information on numeric editing, see "Editing" on page 204.

Examples of ES Editing

```
Value          Format      Output
31415.9        ES12.5      b3.14159E+04
14142.5D+3     ES15.5E4    bb1.41425E+0007
31415.9D-22    ES15.5E1    ***************
                       (with -qxlf90=signedzero)  (with -qxlf90=nosignedzero)
-0.001         ES9.2        -1.00E-03                    -1.00E-03
```

F (Real without Exponent) Editing

Form:
F*w.d*

The **F** edit descriptor directs editing between real and complex numbers in internal form and their character representations without exponents.

The **F** edit descriptor can correspond to an input/output list item of type real, to either part (real or imaginary) of an input/output list item of type complex, or to any other type, so long as the length is at least 4 bytes.

The input field for the **F** edit descriptor consists of, in order:

1. An optional sign.
2. A string of digits optionally containing a decimal point. If the decimal point is present, it overrides the *d* specified in the edit descriptor. If the

decimal point is omitted, the rightmost *d* digits of the string are interpreted as following the decimal point, and leading blanks are converted to zeros if necessary.
3. Optionally, an exponent, having one of the following forms:
   - A signed digit string
   - **E**, **D**, or **Q** followed by zero or more blanks and by an optionally signed digit string. **E**, **D**, and **Q** are processed identically.

The output field for the **F** edit descriptor consists of, in order:
1. Blanks, if necessary.
2. A minus sign if the internal value is negative, or an optional plus sign if the internal value is zero or positive.
3. A string of digits that contains a decimal point and represents the magnitude of the internal value, as modified by the scale factor in effect and rounded to *d* fractional digits. See "P (Scale Factor) Editing" on page 224 for more information.

See also the general information about numeric editing on page 204.

On input, *w* must be greater than zero.

In Fortran 95 on output, w can be zero. If w is zero, the output field consists of the least number of characters required to represent the output value.

Examples of F Editing on Input

(Assume **BN** editing is in effect for blank interpretation.)

```
Input      Format     Value
-100       F6.2       -1.0
2.9        F6.2       2.9
4.E+2      F6.2       400.0
```

Examples of F Editing on Output

```
Value      Format        Output                     Output
                   (with -qxlf77=noleadzero)  (with -qxlf77=leadzero)
+1.2       F8.4         bb1.2000                   bb1.2000
·12345     F8.3         bbbb.123                   bbbb0.123
-12.34     F6.2         -12.34                     -12.34
-12.34     F0.2         -12.34                     -12.34

                   (with -qxlf90=signedzero)  (with -qxlf90=nosignedzero)
-0.001     F5.2         -0.00                       b0.00
```

G (General) Editing

Forms:

G*w*.*d*
G*w*.*d***E***e*
G*w*.*d***D***e*

G*w***.***d***Q***e*

The **G** edit descriptor can correspond to an input/output list item of any type. Editing of integer data follows the rules of the **I** edit descriptor; editing of real and complex data follows the rules of the **E** or **F** edit descriptors (depending on the magnitude of the value); editing of logical data follows the rules of the **L** edit descriptor; and editing of character data follows the rules of the **A** edit descriptor.

Generalized Real and Complex Editing

If the **nogedit77** suboption (the default) of the **-qxlf77** option is specified, the method of representation in the output field depends on the magnitude of the datum being edited. Let N be the magnitude of the internal datum. If $0 < N < 0.1-0.5\times10^{-d-1}$ or $N \geq 10^{d}-0.5$ or N is 0 and d is 0, **G***w.d* output editing is the same as *k***PE** *w.d* output editing and **G***w.d***E***e* output editing is the same as *k***PE***w.d***E***e* output editing, where *k***P** refers to the scale factor ("P (Scale Factor) Editing" on page 224) currently in effect. If $0.1-0.5\times10^{-d-1} \leq N < 10^{d}-0.5$ or N is identically 0 and d is not zero, the scale factor has no effect, and the value of N determines the editing as follows:

Magnitude of Datum	Equivalent Conversion
$N = 0$	F(*w–n*).(*d–1*),*n*('*b*') (d must not be 0)
$0.1-0.5\times10^{-d-1} \leq N < 1-0.5\times10^{-d}$	F(*w–n*).*d*,*n*('*b*')
$1-0.5\times10^{-d} \leq N < 10-0.5\times10^{-d+1}$	F(*w–n*).(*d–1*),*n*('*b*')
$10-0.5\times10^{-d+1} \leq N < 100-0.5\times10^{-d+2}$	F(*w–n*).(*d–2*),*n*('*b*')
...	...
$10^{d-2}-0.5\times10^{-2} \leq N < 10^{d-1}-0.5\times10^{-1}$	F(*w–n*).1,*n*('*b*')
$10^{d-1}-0.5\times10^{-1} \leq N < 10^{d}-0.5$	F(*w–n*).0,*n*('*b*')

where *b* is a blank. *n* is 4 for **G***w.d* and *e*+2 for **G***w.d***E***e*. The value of *w-n* must also be positive.

Note that the scale factor has no effect unless the magnitude of the datum to be edited is outside the range that permits effective use of F editing.

If $0 < N < 0.1-0.5\times10^{-d-1}$, $N \geq 10^{d}-0.5$, or N is 0 and d is 0, **G***w.d***D***e* output editing is the same as *k***PE***w.d***D***e* output editing and **G***w.d***Q***e* output editing is the same as *k***PE***w.d***Q***e* output editing.

On output, if the **gedit77** suboption of the **-qxlf77** compiler option is specified, the number is converted using either **E** or **F** editing, depending on the number. The field is padded with blanks on the right as necessary. Letting *N* be the magnitude of the number, editing is as follows:

- If $N<0.1$ or $N \geq 10^d$:
  - **G***w.d* editing is the same as **E***w.d* editing
  - **G***w.d***E***e* editing is the same as **E***w.d***E***e* editing.
- If $N \geq 0.1$ and $N<10^d$:

Magnitude of Datum	Equivalent Conversion
$0.1 \leq N < 1$	F(*w*-*n*).d, *n*('*b*')
$1 \leq N < 10$	F(*w*-*n*).(d-1), *n*('*b*')
.	.
.	.
$10^{d-2} \leq N < 10^{d-1}$	F(*w*-*n*).1, *n*('*b*')
$10^{d-1} \leq N < 10^d$	F(*w*-*n*).0, *n*('*b*')

Note: While FORTRAN 77 does not address how rounding of values affects the output field form, Fortran 90 does. Therefore, using **-qxlf77=gedit77** may produce a different output form than **-qxlf77=nogedit77** for certain combinations of values and **G** edit descriptors.

See the general information about numeric editing on page 204.

See “Editing” on page 204 for additional information.

Examples of G Editing on Output

```
Value       Format         Output                     Output
                     (with -qxlf77=gedit77)    (with -qxlf77=nogedit77)
0.0         G10.2        bb0.00E+00                 bbb0.0
0.0995      G10.2        bb0.10E+00                 bb0.10
99.5        G10.2        bb100.                     bb0.10E+03
```

I (Integer) Editing

Forms:

I*w*
I*w.m*

The **I** edit descriptor directs editing between integers in internal form and character representations of integers. The corresponding input/output list item can be of type integer or any other type.

w includes the optional sign.

m must have a value that is less than or equal to *w*, unless *w* is zero.

The input field for the **I** edit descriptor must be an optionally signed digit string, unless it is all blanks. If it is all blanks, the input field is considered to be zeros.

m is useful on output only. It has no effect on input.

On input, *w* must be greater than zero.

On output, *w* can be zero. If *w* is zero, the output field consists of the least number of characters required to represent the output value.

The output field for the **I** edit descriptor consists of, in order:

1. Zero or more leading blanks
2. A minus sign, if the internal value is negative, or an optional plus sign, if the internal value is zero or positive
3. The magnitude in the form of:
   - A digit string without leading zeros if *m* is not specified
   - A digit string of at least *m* digits if *m* is specified and, if necessary, with leading zeros. If the internal value and *m* are both zero, blanks are written.

For additional information about numeric editing, see page 204.

If *m* is zero, *w* is positive and the value of the internal datum is zero, the output field consists of *w* blank characters. If both *w* and *m* are zero and the value of the internal datum is zero, the output field consists of only one blank character.

Examples of I Editing on Input

(Assume **BN** editing is in effect for blank interpretation.)

```
Input      Format     Value
-123       I6         -123
123456     I7.5       123456
1234       I4         1234
```

Examples of I Editing on Output

```
Value      Format     Output
-12        I7.6       -000012
12345      I5         12345
0          I6.0       bbbbbb
0          I0.0       b
2          I0         2
```

L (Logical) Editing

Form:

L*w*

The **L** edit descriptor directs editing between logical values in internal form and their character representations. The **L** edit descriptor can correspond to an input/output list item of type logical, or any other type.

The input field consists of optional blanks, followed by an optional decimal point, followed by a `T` for true or an `F` for false. *w* includes blanks. Any

characters following the T or F are accepted on input but are ignored; therefore, the strings .TRUE. and .FALSE. are acceptable input forms.

The output field consists of T or F preceded by (*w* - 1) blanks.

Examples of L Editing on Input

```
Input      Format     Value
T          L4         true
·FALSE.    L7         false
```

Examples of L Editing on Output

```
Value      Format     Output
TRUE       L4         bbbT
FALSE      L1         F
```

O (Octal) Editing

Forms:
O*w*
O*w*.*m*

The **O** edit descriptor directs editing between values of any type in internal form and their octal representation. (An octal digit is one of 0-7.)

w includes blanks.

On input, *w* octal digits are edited and form the internal representation for the value of the input list item. The octal digits in the input field correspond to the rightmost octal digits of the internal representation of the value assigned to the input list item. *m* has no effect on input.

On input, *w* must be greater than zero.

On output, *w* can be zero. If *w* is zero, the output field consists of the least number of characters required to represent the output value.

The output field for **O***w* consists of zero or more leading blanks followed by the internal value in a form identical to the octal digits without leading zeros. Note that an octal constant always consists of at least one digit.

The output field for **O***w*.*m* is the same as for **O***w*, except that the digit string consists of at least *m* digits. If necessary, the digit string is padded with leading zeros. The value of *m* must not exceed the value of *w*, unless *w* is zero. If *m* is zero and the value of the internal datum is zero, the output field consists of only blank characters, regardless of the sign control in effect.

If the **nooldboz** suboption of the **-qxlf77** compiler option is specified (the default), asterisks are printed when the output field width is not sufficient to contain the entire output. On input, the **BN** and **BZ** edit descriptors affect the **O** edit descriptor.

If the **oldboz** suboption of the **-qxlf77** compiler option is specified, the following occurs on output:

- **O***w* is treated as **O***w.m*, with *m* assuming the value that is the minimum of *w* and the number of digits required to represent the maximum possible value of the data item.
- The output consists of blanks followed by at least *m* digits. These are the rightmost digits of the number, zero-filled if necessary, until there are *m* digits. If the number is too large to fit into the output field, only the rightmost *m* digits are output.

If *w* is zero, the **oldboz** suboption will be ignored.

With the **oldboz** suboption, the **BN** and **BZ** edit descriptors do not affect the **O** edit descriptor.

If *m* is zero, *w* is positive and the value of the internal datum is zero, the output field consists of *w* blank characters. If both *w* and *m* are zero and the value of the internal datum is zero, the output field consists of only one blank character.

Examples of O Editing on Input

```
Input      Format      Value
123        O3          83
120        O3          80
```

Examples of O Editing on Output

```
Value      Format        Output                 Output
                  (with -qxlf77=oldboz)  (with -qxlf77=nooldboz)
80         O5            00120                  bb120
83         O2            23                     **
0          O5.0          bbbbb                  bbbbb
0          O0.0          b                      b
80         O0            120                    120
```

Q (Character Count) Editing

Form:

Q

The character count **Q** edit descriptor returns the number of characters remaining in an input record. The result can be used to control the rest of the input.

There also exists the extended precision **Q** edit descriptor. By default, XL Fortran only recognizes the extended precision **Q** edit descriptor described earlier on page 207. To enable both **Q** edit descriptors, you must specify the **-qqcount** compiler option. See "-qqcount Option" in the *User's Guide* for more information.

When you specify the **-qqcount** compiler option, the compiler will distinguish between the two **Q** edit descriptors by the way the **Q** edit descriptor is used. If only a solitary **Q** is found, the compiler will interpret it as the character count **Q** edit descriptor. If **Q***w.* or **Q***w.d* is encountered, XL Fortran will interpret it as the extended precision **Q** edit descriptor. You should use correct format specifications with the proper separators to ensure that XL Fortran correctly interprets which **Q** edit descriptor you specified.

The value returned as a result of the character count **Q** edit descriptor depends on the length of the input record and on the current character position in that record. The value is returned into a scalar integer variable on the **READ** statement whose position corresponds to the position of the character count **Q** edit descriptor in the **FORMAT** statement.

The character count **Q** edit descriptor can read records of the following file types and access modes:
- Formatted sequential external files. A record of this file type is terminated by a new-line character. Records in the same file have different lengths.
- Formatted sequential internal nonarray files. The record length is the length of the scalar character variable.
- Formatted sequential internal array files. The record length is the length of an element in the character array.
- Formatted direct external files. The record length is the length specified by the **RECL=** specifier in the **OPEN** statement.

In an output operation, the character count **Q** edit descriptor is ignored. The corresponding output item is skipped.

Examples of Character Count Q Editing on Input

```
@PROCESS QCOUNT
       CHARACTER(50) BUF
       INTEGER(4) NBYTES
       CHARACTER(60) STRING
       ...
       BUF = 'This string is 29 bytes long.'
       READ( BUF, FMT='(Q)' ) NBYTES
       WRITE( *,* ) NBYTES
! NBYTES equals 50 because the buffer BUF is 50 bytes long.
       READ(*,20) NBYTES, STRING
20     FORMAT(Q,A)
! NBYTES will equal the number of characters entered by the user.
       END
```

Z (Hexadecimal) Editing

Forms:

Z*w*

Z*w***.***m*

The **Z** edit descriptor directs editing between values of any type in internal form and their hexadecimal representation. (A hexadecimal digit is one of 0-9, A-F, or a-f.)

On input, *w* hexadecimal digits are edited and form the internal representation for the value of the input list item. The hexadecimal digits in the input field correspond to the rightmost hexadecimal digits of the internal representation of the value assigned to the input list item. *m* has no effect on input.

On output, *w* can be zero. If *w* is zero, the output field consists of the least number of characters required to represent the output value.

The output field for **Z***w* consists of zero or more leading blanks followed by the internal value in a form identical to the hexadecimal digits without leading zeros. Note that a hexadecimal constant always consists of at least one digit.

The output field for **Z***w.m* is the same as for **Z***w*, except that the digit string consists of at least *m* digits. If necessary, the digit string is padded with leading zeros. The value of *m* must not exceed the value of *w*, unless *w* is zero. If *m* is zero and the value of the internal datum is zero, the output field consists of only blank characters, regardless of the sign control in effect.

If *m* is zero, *w* is positive and the value of the internal datum is zero, the output field consists of *w* blank characters. If both *w* and *m* are zero and the value of the internal datum is zero, the output field consists of only one blank character.

If the **nooldboz** suboption of the **-qxlf77** compiler option is specified (the default), asterisks are printed when the output field width is not sufficient to contain the entire output. On input, the **BN** and **BZ** edit descriptors affect the **Z** edit descriptor.

If the **oldboz** suboption of the **-qxlf77** compiler option is specified, the following occurs on output:

- **Z***w* is treated as **Z***w.m*, with *m* assuming the value that is the minimum of *w* and the number of digits required to represent the maximum possible value of the data item.

- The output consists of blanks followed by at least *m* digits. These are the rightmost digits of the number, zero-filled if necessary, until there are *m* digits. If the number is too large to fit into the output field, only the rightmost *m* digits are output.

If *w* is zero, the **oldboz** suboption will be ignored.

With the **oldboz** suboption, the **BN** and **BZ** edit descriptors do not affect the **Z** edit descriptor.

Examples of Z Editing on Input

```
Input     Format     Value
0C        Z2         12
7FFF      Z4         32767
```

Examples of Z Editing on Output

```
Value     Format        Output                   Output
                  (with -qxlf77=oldboz)  (with -qxlf77=nooldboz)
-1        Z2            FF                       **
12        Z4            000C                     bbbC
12        Z0            C                        C
0         Z5.0          bbbbb                    bbbbb
0         Z0.0          b                        b
```

Control Edit Descriptors

/ (Slash) Editing

Forms:

```
/
r/
```

The slash edit descriptor indicates the end of data transfer on the current record. The repeat specifier (*r*) has a default value of 1.

When you connect a file for input using sequential access, each slash edit descriptor positions the file at the beginning of the next record.

When you connect a file for output using sequential access, each slash edit descriptor creates a new record and positions the file to write at the start of the new record.

When you connect a file for input or output using direct access, each slash edit descriptor increases the record number by one, and positions the file at the beginning of the record that has that record number.

Examples of Slash Editing on Input

```
500   FORMAT(F6.2 / 2F6.2)
100   FORMAT(3/)
```

: (Colon) Editing

Form:

:

The colon edit descriptor terminates format control (which is discussed on page 226) if no more items are in the input/output list. If more items are in the input/output list when the colon is encountered, it is ignored.

Example of Colon Editing

```
10    FORMAT(3(:'Array Value',F10.5)/)
```

$ (Dollar) Editing

Form:

`$`

The dollar edit descriptor inhibits an end-of-record for a sequential **WRITE** statement. Usually, when the end of a format specification is reached, data transmission of the current record ceases and the file is positioned so that the next input/output operation processes a new record. But, if a dollar sign occurs in the format specification, the automatic end-of-record action is suppressed. Subsequent input/output statements can continue writing to the same record.

Example of Dollar Editing

A common use for dollar sign editing is to prompt for a response and read the answer from the same line.

```
      WRITE(*,FMT='($,A)')'Enter your age  '
      READ(*,FMT='(BN,I3)')IAGE
      WRITE(*,FMT=1000)
1000  FORMAT('Enter your height:  ',$)
      READ(*,FMT='(F6.2)')HEIGHT
```

Apostrophe/Double Quotation Mark Editing (Character-String Edit Descriptor)

Forms:

'*character string*'
"*character string*"

The apostrophe/double quotation mark edit descriptor specifies a character literal constant in an output format specification. The width of the output field is the length of the character literal constant. See page 35 for additional information on character literal constants.

Notes:

1. A backslash is recognized as an escape sequence by default, and as a backslash character when the **-qnoescape** compiler option is specified. See page 36 for more information on escape sequences.
2. XL Fortran provides support for multibyte characters within character constants, Hollerith constants, character-string edit descriptors, and comments. This support is provided through the **-qmbcs** option. Assignment of a constant containing multibyte characters to a variable that is not large enough to hold the entire string may result in truncation within a multibyte character.
3. Support is also provided for Unicode characters and filenames. If the environment variable **LANG** is set to **UNIVERSAL** and the **-qmbcs** compiler option is specified, the compiler can read and write Unicode characters and filenames.

Examples of Apostrophe/Double Quotation Mark Editing

```
      ITIME=8
      WRITE(*,5) ITIME
5     FORMAT('The value is -- ',I2)          ! The value is -- 8
      WRITE(*,10) ITIME
10    FORMAT(I2,'o''clock')                 ! 8o'clock
      WRITE(*,'(I2,''o''''clock'')') ITIME    ! 8o'clock
      WRITE(*,15) ITIME
15    FORMAT("The value is -- ",I2)          ! The value is -- 8
      WRITE(*,20) ITIME
20    FORMAT(I2,"o'clock")                   ! 8o'clock
      WRITE(*,'(I2,"o''clock")') ITIME       ! 8o'clock
```

BN (Blank Null) and BZ (Blank Zero) Editing

Forms:
BN
BZ

The **BN** and **BZ** edit descriptors control the interpretation of nonleading blanks by subsequently processed **I**, **F**, **E**, **EN**, **ES**, **D**, **G**, **B**, **O**, **Z**, and extended precision **Q** edit descriptors. **BN** and **BZ** have effect only on input.

BN specifies that blanks in numeric input fields are to be ignored, and remaining characters are to be interpreted as though they were right-justified. A field of all blanks has a value of zero.

BZ specifies that nonleading blanks in numeric input fields are to be interpreted as zeros.

The initial setting for blank interpretation is determined by the **BLANK=** specifier of the **OPEN** statement. (See "OPEN" on page 355.) The initial setting is determined as follows:

- If **BLANK=** is not specified, blank interpretation is the same as if **BN** editing were specified.
- If **BLANK=** is specified, blank interpretation is the same as if **BN** editing were specified when the specifier value is **NULL**, or the same as if **BZ** editing were specified when the specifier value is **ZERO**.

The initial setting for blank interpretation takes effect at the start of a formatted **READ** statement and stays in effect until a **BN** or **BZ** edit descriptor is encountered or until format control finishes. Whenever a **BN** or **BZ** edit descriptor is encountered, the new setting stays in effect until another **BN** or **BZ** edit descriptor is encountered, or until format control terminates.

If you specify the **oldboz** suboption of the **xlf77** compiler option, the **BN** and **BZ** edit descriptors do not affect data input edited with the **B**, **O**, or **Z** edit descriptors. Blanks are interpreted as zeros.

H Editing

Form:
*n***H** *str*

The **H** edit descriptor specifies a character string (*str*) and its length (*n*) in an output format specification. The string can consist of any of the characters allowed in a character literal constant.

If an **H** edit descriptor occurs within a character literal constant, the constant delimiter character (for example, apostrophe) can be represented within *str* if two such characters are consecutive. Otherwise, another delimiter must be used.

The **H** edit descriptor must not be used on input.

Notes:

1. A backslash is recognized,as an escape character by default, and as a backslash character when the **-qnoescape** compiler option is specified. See page 36 for more information on escape sequences.
2. XL Fortran provides support for multibyte characters within character constants, Hollerith constants, character-string edit descriptors, and comments. This support is provided through the **-qmbcs** option. Assignment of a constant containing multibyte characters to a variable that is not large enough to hold the entire string may result in truncation within a multibyte character.
3. Support is also provided for Unicode characters and filenames. If the environment variable **LANG** is set to **UNIVERSAL** and the **-qmbcs** compiler option is specified, the compiler can read and write Unicode characters and filenames.

4. Fortran 95 does not include the **H** edit descriptor, although it was part of both FORTRAN 77 and Fortran 90. See page "Deleted Features" on page 735 for more information.

Examples of H Editing

```
50    FORMAT(16HThe value is -- ,I2)
10    FORMAT(I2,7Ho'clock)
      WRITE(*,'(I2,7Ho'clock)') ITIME
```

P (Scale Factor) Editing

Form:
*k***P**

The scale factor, *k*, applies to all subsequently processed **F**, **E**, **EN**, **ES**, **D**, **G**, and extended precision **Q** edit descriptors until another scale factor is encountered or until format control terminates. The value of *k* is zero at the beginning of each input/output statement. It is an optionally signed integer value representing a power of ten.

On input, when an input field using an **F**, **E**, **EN**, **ES**, **D**, **G**, or extended precision **Q** edit descriptor contains an exponent, the scale factor is ignored. Otherwise, the internal value equals the external value multiplied by $10^{(-k)}$.

On output:

- In **F** editing, the external value equals the internal value multiplied by 10^{k}.
- In **E**, **D**, and extended precision **Q** editing, the external decimal field is multiplied by 10^{k}. The exponent is then reduced by *k*.
- In **G** editing, fields are not affected by the scale factor unless they are outside the range that can use **F** editing. If the use of **E** editing is required, the scale factor has the same effect as with **E** output editing.
- In **EN** and **ES** editing, the scale factor has no effect.

Examples of P Editing on Input

```
Input       Format      Value
98.765      3P,F8.6     .98765E-1
98.765      -3P,F8.6    98765.
.98765E+2   3P,F10.5    .98765E+2
```

Examples of P Editing on Output

```
Value      Format             Output                     Output
                    (with -qxlf77=noleadzero)  (with -qxlf77=leadzero)
5.67       -3P,F7.2          bbbb.01                    bbb0.01
12.34      -2P,F6.4          b.1234                     0.1234
12.34      2P,E10.3          b12.34E+00                 b12.34E+00
```

S, SP, and SS (Sign Control) Editing

Forms:
S

SP
SS

The **S**, **SP**, and **SS** edit descriptors control the output of plus signs by all subsequently processed **I**, **F**, **E**, **EN**, **ES**, **D**, **G**, and extended precision **Q** edit descriptors until another **S**, **SP**, or **SS** edit descriptor is encountered or until format control terminates.

S and **SS** specify that plus signs are not to be written. (They produce identical results.) **SP** specifies that plus signs are to be written.

Examples of S, SS, and SP Editing on Output

```
Value      Format     Output
12.3456    S,F8.4     b12.3456
12.3456    SS,F8.4    b12.3456
12.3456    SP,F8.4    +12.3456
```

T, TL, TR, and X (Positional) Editing

Forms:
T*c*
TL*c*
TR*c*
*o***X**

The **T**, **TL**, **TR**, and **X** edit descriptors specify the position where the transfer of the next character to or from a record starts. This position is:
- For **T***c*, the *c*th character position.
- For **TL***c*, *c* characters backward from the current position. If the value of *c* is greater than or equal to the current position, the next character accessed is position 1 of the record.
- For **TR***c*, *c* characters forward from the current position.
- For *o***X**, *o* characters forward from the current position.

The **TR** and **X** edit descriptors give identical results.

On input, a **TR** or **X** edit descriptor can specify a position beyond the last character of the record if no characters are transferred from that position.

On output, a **T**, **TL**, **TR**, or **X** edit descriptor does not by itself cause characters to be transferred. If characters are transferred to positions at or after the position specified by the edit descriptor, positions skipped and previously unfilled are filled with blanks. The result is the same as if the entire record were initially filled with blanks.

On output, a **T**, **TL**, **TR**, or **X** edit descriptor can result in repositioning so that subsequent editing with other edit descriptors causes character replacement.

The **X** edit descriptor can be specified without a character position. It is treated as 1X. When the source file is compiled with **-qlanglvl=90std** or **-qlanglvl=95std**, this extension is disabled in all compile-time format specifications, and the form of *o***X** is enforced. To disable this extension in run-time formats, the following run-time option must be set:

```
XLFRTEOPTS="langlvl=90std" or "langlvl=95std" ; export XLFRTEOPTS
```

Examples of T, TL, and X Editing on Input

```
150   FORMAT(I4,T30,I4)
200   FORMAT(F6.2,5X,5(I4,TL4))
```

Examples of T, TL, TR, and X Editing on Output

```
50    FORMAT('Column 1',5X,'Column 14',TR2,'Column 25')
100   FORMAT('aaaaa',TL2,'bbbbb',5X,'ccccc',T10,'ddddd')
```

Interaction between Input/Output Lists and Format Specifications

The beginning of format-directed formatting initiates format control. Each action of format control depends on the next edit descriptor contained in the format specification and on the next item in the input/output list, if one exists.

If an input/output list specifies at least one item, at least one data edit descriptor must exist in the format specification. Note that an empty format specification (parentheses only) can be used only if there are no items in the input/output list or if each item is a zero-sized array. If this is the case and advancing input/output is in effect, one input record is skipped, or one output record containing no characters is written. For nonadvancing input/output, the file position is left unchanged.

A format specification is interpreted from left to right, except when a repeat specification (*r*) is present. A format item that is preceded by a repeat specification is processed as a list of *r* format specifications or edit descriptors identical to the format specification or edit descriptor without the repeat specification.

One item specified by the input/output list corresponds to each data edit descriptor. A list item of type complex requires the interpretation of two **F**, **E**, **EN**, **ES**, **D**, **G**, or extended precision **Q** edit descriptors. No item specified by the input/output list corresponds to a control edit descriptor or character string edit descriptor. Format control communicates information directly with the record.

Format control operates as follows:

1. If a data edit descriptor is encountered, format control processes an input/output list item, if there is one, or terminates the input/output

command if the list is empty. If the list item processed is of type complex, any two edit descriptors are processed.

2. The colon edit descriptor terminates format control if no more items are in the input/output list. If more items are in the input/output list when the colon is encountered, it is ignored.
3. If the end of the format specification is reached, format control terminates if the entire input/output list has been processed, or control reverts to the beginning of the format item terminated by the last preceding right parenthesis. The following items apply when the latter occurs:
   - The reused portion of the format specification must contain at least one data edit descriptor.
   - If reversion is to a parenthesis that is preceded by a repeat specification, the repeat specification is reused.
   - Reversion, of itself, has no effect on the scale factor, on the **S**, **SP**, or **SS** edit descriptors, or on the **BN** or **BZ** edit descriptors.
   - If format control reverts, the file is positioned in a manner identical to the way it is positioned when a slash edit descriptor is processed.

During a read operation, any unprocessed characters of the record are skipped whenever the next record is read. A comma can be used as a value separator for noncharacter data in an input record processed under format-directed formatting. The comma will override the format width specifications when the comma appears before the end of the field width. For example, the format (`I10,F20.10,I4`) will read the following record correctly:

```
-345, .05E-3, 12
```

It is important to consider the maximum size record allowed on the input/output medium when defining a Fortran record by a **FORMAT** statement. For example, if a Fortran record is to be printed, the record should not be longer than the printer's line length.

List-Directed Formatting

In list-directed formatting, editing is controlled by the types and lengths of the data being read or written. An asterisk format identifier specifies list-directed formatting. For example:

```
REAL TOTAL1, TOTAL2
PRINT *, TOTAL1, TOTAL2
```

List-directed formatting can only be used with sequential files.

The characters in a formatted record processed under list-directed formatting constitute a sequence of values separated by value separators:
- A value has the form of a constant or null value.

- A value separator is a comma, slash, or set of contiguous blanks. A comma or slash can be preceded and followed by one or more blanks.

List-Directed Input

Input list items in a list-directed **READ** statement are defined by corresponding values in records. The form of each input value must be acceptable for the type of the input list item. An input value has one of the following forms:
- *c*
- *r* * *c*
- *r* *

c is a literal constant of intrinsic type or a non-delimited character constant. *r* is an unsigned, nonzero, integer literal constant. A kind type parameter must not be specified for either *r* or *c*. The constant *c* is interpreted as though it had the same kind type parameter as the corresponding list item.

The *r* * *c* form is equivalent to *r* successive appearances of the constant. The *r* * form is equivalent to *r* successive appearances of the null value.

A null value is represented by one of the following:
- Two successive commas, with zero or more intervening blanks
- A comma followed by a slash, with zero or more intervening blanks
- An initial comma in the record, preceded by zero or more blanks

Use the **-qintlog** compiler option to specify integer or logical values for input items of either integer or logical type.

A character value can be continued in as many records as required. If the next effective item is of type character and the following are true:
1. The character constant does not contain the value separators blank, comma, or slash, and
2. The character constant does not cross a record boundary, and
3. The first nonblank character is not a quotation mark or apostrophe, and
4. The leading characters are not numeric followed by an asterisk, and
5. The character constant contains at least one character,

the delimiting apostrophes or quotation marks are not required. If the delimiters are omitted, the character constant is terminated by the first blank, comma, slash, or end-of-record, and apostrophes and double quotation marks within the datum are not to be doubled.

The end of a record:
- Has the same effect as a blank separator, unless the blank is within a character literal constant or complex literal constant

- Does not cause insertion of a blank or any other character in a character value
- Must not separate two apostrophes representing an apostrophe.

Two or more consecutive blanks are treated as a single blank unless the blanks are within a character value.

A null value has no effect on the definition status of the corresponding input list item.

A slash indicates the end of the input list, and list-directed formatting is terminated. If additional items remain in the input list when a slash is encountered, it is as if null values had been specified for those items.

If an object of derived type occurs in an input list, it is treated as if all the structure components were listed in the same order as in the definition of the derived type.

List-Directed Output

List-directed **WRITE** and **PRINT** statements produce values in the order they appear in an output list. Values are written in a form that is valid for the data type of each output list item.

Except for complex constants and character constants, the end of a record must not occur within a constant and blanks must not appear within a constant.

Integer values are written using **I** editing.

Real values are written using **E** or **F** editing. (See "E, D, and Q (Extended Precision) Editing" on page 207 or "F (Real without Exponent) Editing" on page 211 for more information.)

Complex constants are enclosed in parentheses with a comma separating the real and imaginary parts, each produced as defined above for real constants. The end of a record can occur between the comma and the imaginary part only if the entire constant is as long as (or longer than) an entire record. The only embedded blanks permitted within a complex constant are one blank between the comma and the end of a record, and one blank at the beginning of the next record.

Logical values are written as `T` for the value true and `F` for the value false.

Character constants produced for an internal file, or for a file opened without a **DELIM=** specifier or with a **DELIM=** specifier with a value of **NONE**:
- Are not delimited by apostrophes or quotation marks,

- Are not separated from each other by value separators,
- Have each internal apostrophe or double quotation mark represented externally by one apostrophe or double quotation mark, and
- Have a blank character inserted by the processor for carriage control at the beginning of any record that begins with the continuation of a character constant from the preceding record.

Undelimited character data may not be read back correctly using list-directed input.

Character constants produced for a file opened with a **DELIM=** specifier with a value of **QUOTE** are delimited by double quotation marks, followed by a value separator, and have each internal quote represented on the external medium by two contiguous double quotation marks. Character constants produced for a file opened with a **DELIM=** specifier with a value of **APOSTROPHE** are delimited by apostrophes, followed by a value separator, and have each internal apostrophe represented on the external medium by two contiguous apostrophes.

Slashes (as value separators) and null values are not written.

Arrays are written in column-major order.

You can specify a structure in an output list. On list-directed output, a structure is treated as if all of its components were listed in the same order as they are defined in the derived-type definition.

The following table shows the width of the written field for any data type and length. The size of the record will be the sum of the field widths plus a byte to separate each noncharacter field.

Table 13. Width of Written Field

Data Type	Length (bytes)	Maximum Field Width (characters)	Fraction (decimal digits)	Precision/IEEE (decimal digits)
integer	1 2 4 8	4 6 11 20	n/a n/a n/a n/a	n/a n/a n/a n/a
real	4 8 16	17 26 43	10 18 35	7 15 31
complex	8 16 32	37 55 89	10 18 35	7 15 31

Table 13. Width of Written Field (continued)

Data Type	Length (bytes)	Maximum Field Width (characters)	Fraction (decimal digits)	Precision/IEEE (decimal digits)
logical	1 2 4 8	1 1 1 1	n/a n/a n/a n/a	n/a n/a n/a n/a
character	n	n	n/a	n/a

Except for continuation of delimited character constants, each output record begins with a blank character to provide carriage control when the record is printed.

Namelist Formatting

In Fortran 90, namelist formatting can only be used with sequential files.

XLF also allows namelist formatting to be used with internal files.

Namelist Input Data

The form of input for namelist input is:

1. Optional blanks
2. The ampersand (&) character, followed immediately by the namelist group name specified in the **NAMELIST** statement
3. One or more blanks
4. A sequence of zero or more name-value subsequences, separated by value separators
5. A slash to terminate the namelist input

Blanks at the beginning of an input record that continues a delimited character constant are considered part of the constant.

If the **NAMELIST** run-time option has the value **OLD**, input for a **NAMELIST** statement consists of:

1. Optional blanks
2. An ampersand (&) or dollar sign ($), followed immediately by the namelist group name specified in the **NAMELIST** statement
3. One or more blanks
4. A sequence of zero or more name-value subsequences separated from each other by a single comma. A comma may be specified after the last name-value subsequence.
5. `&END` or `$END` to signal the end of the data group
6. The first character of each input record must be blank, including those records that continue a delimited character constant.

In Fortran 95, comments can be used in namelists.

Depending on whether a value of **NEW** or **OLD** is specified for the **NAMELIST** runtime option, different rules apply.

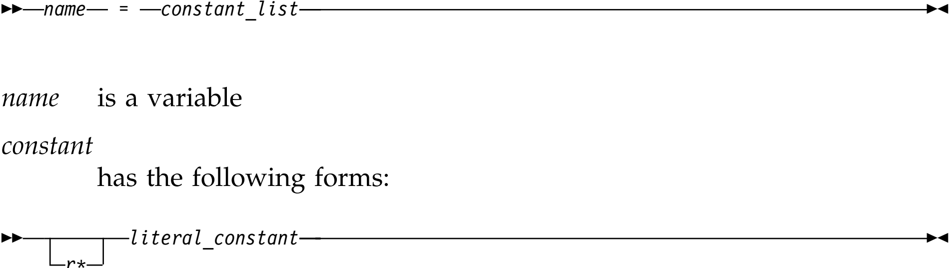
If a value of **NEW** is specified for the **NAMELIST** runtime option, the rules for namelist comments are:

- Except within a character literal constant, an exclamation point (!) after a value separator, except a slash, or in the first nonblank position of a namelist input record initiates a comment.
- The comment extends to the end of the input record, and can contain any character in the processor-dependent character set.
- The comment is ignored.
- A slash within a namelist comment does not terminate execution of the namelist input statement.

If a value of **OLD** is specified for the **NAMELIST** runtime option, the rules for namelist comments are:

- Except within a character literal constant, an exclamation point (!) after a single comma or in the first nonblank position of a namelist input record, but not the first character of an input record, initiates a comment.
- The comment extends to the end of the input record, and can contain any character in the processor-dependent character set.
- The comment is ignored.
- A &END or $END within a namelist comment does not terminate execution of the namelist input statement.

The form of a name-value subsequence in an input record is:



name is a variable

constant
has the following forms:

r is an unsigned, nonzero, scalar, integer literal constant specifying the number of times the *literal_constant* is to occur. *r* cannot specify a kind type parameter.

literal_constant
is a scalar literal constant of intrinsic type that cannot specify a kind type parameter, or it is a null value. The constant is treated as if it had the same kind type parameter as the corresponding list item. If *literal_constant* is of type character, it must be delimited by apostrophes or quotation marks. If *literal_constant* is of type logical, it can be specified as T or F.

Any subscripts, strides, and substring range expressions used to qualify *name* must be integer literal constants with no kind type parameter specified.

For information on the type of noncharacter input data, see “List-Directed Input” on page 228.

If *name* is neither an array nor an object of derived type, *constant_list* must contain only a single constant.

Variable names specified in the input file must appear in the namelist list, but the order of the input data is not significant. A name that has been made equivalent to *name* cannot be substituted for that name in the namelist list. See “NAMELIST” on page 353 for details on what can appear in a namelist list.

In each name-value subsequence, the name must be the name of a namelist group item with an optional qualification. The name with the optional qualification must not be a zero-sized array, zero-sized array section, or zero-length character string. The optional qualification, if specified, must not contain a vector subscript.

If *name* is an array or array section without vector subscripts, it is expanded into a list of all the elements of the array, in the order that they are stored. If *name* is a structure, it is expanded into a list of ultimate components of intrinsic type, in the order specified in the derived-type definition.

If *name* is an array or structure, the number of constants in *constant_list* must be less than or equal to the number of items specified by the expansion of *name*. If the number of constants is less than the number of items, the remaining items retain their former values.

A null value is specified by:
- The *r* * form
- Blanks between two consecutive value separators following an equal sign
- Zero or more blanks preceding the first value separator and following an equal sign
- Two consecutive nonblank value separators

A null value has no effect on the definition status of the corresponding input list item. If the namelist group object list item is defined, it retains its previous value; if it is undefined, it remains undefined. A null value must not be used as either the real or imaginary part of a complex constant, but a single null value can represent an entire complex constant.

The end of a record following a value separator, with or without intervening blanks, does not specify a null value.

When the **LANGLVL** run-time option is set to **EXTENDED**, XL Fortran allows multiple input values to be specified in conjunction with a single array element. The array element cannot specify subobject designators. When this occurs, the values are assigned to successive elements of the array, in array element order. For example, suppose that array A is declared as follows:

```
INTEGER A(100)
NAMELIST /FOO/ A
READ (5, FOO)
```

and that the following input appears in unit 5:

```
&FOO
A(3) = 2, 10, 15, 16
/
```

During execution of the **READ** statement, the value 2 is assigned to A(3), 10 is assigned to A(4), 15 is assigned to A(5), and 16 is assigned to A(6).

If multiple values are specified in conjunction with a single array element, any logical constant must be specified with a leading period (for example, .T).

If the **NAMELIST** run-time option is specified with the value **OLD**, the **BLANK=** specifier determines how embedded and trailing blanks between noncharacter constants are treated.

If the **-qmixed** compiler option is specified, the namelist group name and list item names are treated in a case-sensitive manner.

A slash encountered as a value separator during the execution of a namelist input statement causes termination of execution of that input statement after assignment of the previous value. If there are additional items in the namelist group object being transferred, the effect is as if null values had been supplied for them.

Example of Namelist Input Data

File NMLEXP contains the following data before the **READ** statement is executed:

Character position:

```
         1         2         3
1...+....0....+....0....+....0
```

File contents:

```
 &NAME1
 I=5,
 SMITH%P_AGE=40
 /
```

The above file contains four data records. The program contains the following:

```
TYPE PERSON
  INTEGER P_AGE
  CHARACTER(20) P_NAME
END TYPE PERSON
TYPE(PERSON) SMITH
NAMELIST /NAME1/ I,J,K,SMITH
I=1
J=2
K=3
SMITH=PERSON(20,'John Smith')
OPEN(7,FILE='NMLEXP')
READ(7,NML=NAME1)
! Only the value of I and P_AGE in SMITH are
! altered (I = 5, SMITH%P_AGE = 40).
! J, K and P_NAME in SMITH remain the same.
END
```

Note: In the previous example, the data items appear in separate data records. The following example is a file with the same data items, but they are in one data record:

Character position:

```
         1         2         3         4
1...+....0....+....0....+....0....+....0
```

File contents:

```
 &NAME1 I= 5, SMITH%P_AGE=40 /
```

An example of a **NAMELIST** comment when **NAMELIST=NEW** is specified and the **NAMELIST** comment appears after the value separator space.

```
&TODAY I=12345          ! This is a comment. /
X(1)=12345, X(3:4)=2*1.5, I=6,
P="!ISN'T_BOB'S", Z=(123,0)/
```

An example of a **NAMELIST** comment when **NAMELIST=OLD** is specified and the **NAMELIST** comment appears after a comma separator.

```
&TODAY I=12345,        ! This is a comment.
X(1)=12345, X(3:4)=2*1.5, I=6,
P="!ISN'T_BOB'S", Z=(123,0) &END
```

Namelist Output Data

When output data is written using a namelist list, it is written in a form that can be read using a namelist list (except for character data that is not delimited). All variables specified in the namelist list and their values are written out, each according to its type. Character data is delimited as specified by the **DELIM=** specifier. The fields for the data are made large enough to contain all the significant digits. (See Table 13 on page 230 for information on the fields.) The values of a complete array are written out in column-major order.

A **WRITE** statement with a namelist list produces a minimum of three output records: one record containing the namelist name, followed by one or more records containing output data items, and a final record containing the slash (/) end marker. An internal file meant to receive namelist output must be a character array containing at least three elements. More than three array elements may be required, depending on the amount of data transferred in the **WRITE** statement. You cannot use one long character variable, even if it is large enough to hold all of the data. If the length of the array element to hold the data is not sufficient, it will be necessary to specify an array with more than three array elements.

If the **NAMELIST** run-time option is not specified or if **NAMELIST=NEW**, the namelist group name and namelist item names are output in uppercase.

If **NAMELIST=OLD** is specified, the namelist group name and namelist item names are output in lower case. If the **-qmixed** compiler option is specified, the name is case sensitive, regardless of the value of the **NAMELIST** run-time option.

If **NAMELIST=OLD** is specified, the end of the output record will be signaled by `&end`.

If the **NAMELIST** run-time option is specified with the value **OLD** and the **DELIM=** specifier is not specified, character data is delimited by apostrophes. Non-delimited character strings will be delimited by apostrophes and will be separated from each other by commas. Also, blanks will not be added to the beginning of a record that starts with the continuation of a character string from the previous record.

Character constants produced for a file opened without a **DELIM=** specifier or with a **DELIM=** specifier with a value of **NONE**:
- Are not delimited by apostrophes or quotation marks,
- Are not separated from each other by value separators,

- Have each internal apostrophe or quotation mark represented externally by one apostrophe or quotation mark, and
- Have a blank character inserted by the processor for carriage control at the beginning of any record that begins with the continuation of a character constant from the preceding record.

Nondelimited character data that has been written out cannot be read as character data.

For internal files, character constants are written with a value of **APOSTROPHE** for the **DELIM=** specifier.

Character constants produced for a file opened with a **DELIM=** specifier with a value of **QUOTE** are delimited by double quotation marks, are preceded and followed by a value separator, and have each internal quotation mark represented on the external medium by two contiguous quotation marks.

Character constants produced for a file opened with a **DELIM=** specifier with a value of **APOSTROPHE** are delimited by apostrophes, are preceded and followed by a value separator, and have each internal apostrophe represented on the external medium by two contiguous apostrophes.

To restrict namelist output records to a given width, specify the **RECL=** specifier (in the **OPEN** statement) or the **NLWIDTH** run-time option. See the *User's Guide* for information on the **NLWIDTH** run-time option.

Except for continuation of delimited character constants, each output record begins with a blank character to provide carriage control when the record is printed.

For external files, by default, all of the output items appear in a single output record wide enough to contain them. To have the record output on separate lines, use the **RECL=** specifier (in the **OPEN** statement) or the **NLWIDTH** run-time option.

For information on the type of noncharacter output data, see "List-Directed Output" on page 229.

Example of Namelist Output Data

```
TYPE PERSON
  INTEGER P_AGE
  CHARACTER(20) P_NAME
END TYPE PERSON
TYPE(PERSON) SMITH
NAMELIST /NL1/ I,J,C,SMITH
CHARACTER(5) :: C='BACON'
INTEGER I,J
```

```
I=12046
J=12047
SMITH=PERSON(20,'John Smith')
WRITE(6,NL1)
END
```

After execution of the **WRITE** statement with **NAMELIST=NEW**, the output data is:

```
    1         2         3         4
1...+....0....+....0....+....0....+....0
 &NL1
 I=12046, J=12047, C=BACON, SMITH=20, John Smith
 /
```

After execution of the **WRITE** statement with **NAMELIST=OLD**, the output data is:

```
    1         2         3         4
1...+....0....+....0....+....0....+....0
 &nl1
 i=12046, j=12047, c='BACON', smith=20, 'John Smith          '
 &end
```

Part 3. Statements and Directives

This section describes the XL Fortran statements as well as reference information on directives, including the symmetric multiprocessing (SMP) directives.

Part 3 provides information on:

- "Chapter 10. Statements" on page 241
- "Chapter 11. Directives" on page 423

© Copyright IBM Corp. 1990, 2000

Chapter 10. Statements

This chapter provides an alphabetical reference to all XL Fortran statements. The section for each statement is organized to help you readily access the syntax and rules, and points to the structure and uses of the statement in Part 2. Concepts and Elements.

The following table lists the statements, and shows which ones are executable, which ones are *specification_part* statements, and which ones can be used as the terminal statement of a **DO** or **DO WHILE** construct.

Table 14. Statements Table

STATEMENT NAME	EXECUTABLE STATEMENT	SPECIFICATION STATEMENT	TERMINAL STATEMENT
ALLOCATABLE		X	
ALLOCATE	X		X
ASSIGN	X		X
AUTOMATIC		X	
BACKSPACE	X		X
BLOCK DATA			
BYTE		X	
CALL	X		X
CASE	X		
CHARACTER		X	
CLOSE	X		X
COMMON		X	
COMPLEX		X	
CONTAINS			
CONTINUE	X		X
CYCLE	X		
DATA		X	
DEALLOCATE	X		X
Derived Type			
DIMENSION		X	
DO	X		
DO WHILE	X		

© Copyright IBM Corp. 1990, 2000

Table 14. Statements Table (continued)

STATEMENT NAME	EXECUTABLE STATEMENT	SPECIFICATION STATEMENT	TERMINAL STATEMENT
DOUBLE COMPLEX		X	
DOUBLE PRECISION		X	
ELSE	X		
ELSE IF	X		
ELSEWHERE	X		
END	X		
END BLOCK DATA			
END DO	X		X
END IF	X		
END FORALL	X		
END FUNCTION	X		
END INTERFACE		X	
END MODULE			
END PROGRAM	X		
END SELECT	X		
END SUBROUTINE	X		
END TYPE		X	
END WHERE	X		
ENDFILE	X		X
ENTRY		X	
EQUIVALENCE		X	
EXIT	X		
EXTERNAL		X	
FORALL	X		X
FORMAT		X	
FUNCTION			
GO TO (Assigned)	X		
GO TO (Computed)	X		X
GO TO (Unconditional)	X		

Table 14. Statements Table (continued)

STATEMENT NAME	EXECUTABLE STATEMENT	SPECIFICATION STATEMENT	TERMINAL STATEMENT
IF (Block)	X		
IF (Arithmetic)	X		
IF (Logical)	X		X
IMPLICIT		X	
INQUIRE	X		X
INTEGER		X	
INTENT		X	
INTERFACE		X	
INTRINSIC		X	
LOGICAL		X	
MODULE			
MODULE PROCEDURE		X	
NAMELIST		X	
NULLIFY	X		X
OPEN	X		X
OPTIONAL		X	
PARAMETER		X	
PAUSE	X		X
POINTER (Fortran 90)		X	
POINTER (integer)		X	
PRINT	X		X
PRIVATE		X	
PROGRAM			
PUBLIC		X	
READ	X		X
REAL		X	
RETURN	X		
REWIND	X		X
SAVE		X	
SELECT CASE	X		

Table 14. Statements Table (continued)

STATEMENT NAME	EXECUTABLE STATEMENT	SPECIFICATION STATEMENT	TERMINAL STATEMENT
SEQUENCE		X	
Statement Function		X	
STATIC		X	
STOP	X		
SUBROUTINE			
TARGET		X	
TYPE		X	
Type Declaration		X	
USE		X	
VIRTUAL		X	
VOLATILE		X	
WAIT	X		X
WHERE	X		X
WRITE	X		X

Assignment and pointer assignment statements are discussed in "Chapter 5. Expressions and Assignment" on page 85. Both statements are executable and can serve as terminal statements.

Attributes

Each attribute has a corresponding attribute specification statement, and the syntax diagram provided for the attribute illustrates this form. An entity can also acquire this attribute from a type declaration statement or, in some cases, through a default setting. For example, entity `A`, said to have the **PRIVATE** attribute, could have acquired the attribute in any of the following ways:

```
REAL, PRIVATE :: A     ! Type declaration statement
PRIVATE :: A           ! Attribute specification statement

MODULE X
  PRIVATE              ! Default setting
  REAL :: A
END MODULE
```

The following table maps out the compatibility of attributes. An "X" indicates whether an entity can have the attributes indicated both horizontally and vertically.

	ALLOCATABLE	AUTOMATIC	DIMENSION	EXTERNAL	INTENT	INTRINSIC	OPTIONAL	PARAMETER	POINTER	PRIVATE	PUBLIC	SAVE	STATIC	TARGET	VOLATILE
ALLOCATABLE		X	X							X	X	X	X	X	X
AUTOMATIC	X		X						X					X	X
DIMENSION	X	X			X		X	X	X	X	X	X	X	X	X
EXTERNAL							X			X	X				
INTENT			X				X							X	X
INTRINSIC										X	X				
OPTIONAL			X	X	X				X					X	X
PARAMETER			X							X	X				
POINTER		X	X				X			X	X	X	X		X
PRIVATE	X		X	X		X		X	X			X	X	X	X
PUBLIC	X		X	X		X		X	X			X		X	X
SAVE	X		X						X	X	X		X	X	X
STATIC	X		X						X	X		X		X	X
TARGET	X	X	X		X		X			X	X	X	X		X
VOLATILE	X	X	X		X		X		X	X	X	X	X	X	

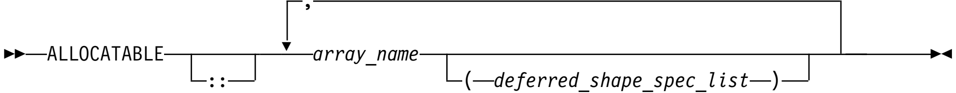
Figure 1.

ALLOCATABLE

Purpose

The **ALLOCATABLE** attribute declares allocatable arrays — that is, arrays whose bounds are determined when space is dynamically allocated by execution of an **ALLOCATE** statement.

Format



array_name
: is the name of an allocatable array

deferred_shape_spec
: is a colon(:), where each colon represents a dimension

Rules

The array cannot be a pointee, a dummy argument, or a function result. If the array is specified elsewhere in the scoping unit with the **DIMENSION** attribute, the array specification must be a *deferred_shape_spec*.

Attributes Compatible with the ALLOCATABLE Attribute

- AUTOMATIC
- DIMENSION
- PRIVATE
- PUBLIC
- SAVE
- STATIC
- TARGET
- VOLATILE

Examples

```
REAL, ALLOCATABLE :: A(:,:)  ! Two-dimensional array A declared
                             ! but no space yet allocated
READ (5,*) I,J
ALLOCATE (A(I,J))
END
```

Related Information

- "Allocatable Arrays" on page 70
- "ALLOCATED(ARRAY)" on page 529
- "ALLOCATE"
- "DEALLOCATE" on page 280
- "Allocation Status" on page 58
- "Deferred-Shape Arrays" on page 69

ALLOCATE

Purpose

The **ALLOCATE** statement dynamically provides storage for pointer targets and allocatable arrays.

Format

```
>>—ALLOCATE—(—allocation_list——————————————————————————)——————————><
                                └,—STAT— = —stat_variable—┘
```

stat_variable
is a scalar integer variable

allocation

allocate_object
is a variable name or structure component. It must be a pointer or an allocatable array.

lower_bound, upper_bound
are each scalar integer expressions

Rules

Execution of an **ALLOCATE** statement for a pointer causes the pointer to become associated with the target allocated. For an allocatable array, the array becomes definable.

The number of dimensions specified (i.e., the number of upper bounds in *allocation*) must be equal to the rank of *allocate_object*. When an **ALLOCATE** statement is executed for an array, the values of the bounds are determined at that time. Subsequent redefinition or undefinition of any entities in the bound expressions does not affect the array specification. Any lower bound, if omitted, is assigned a default value of 1. If any lower bound value exceeds the corresponding upper bound value, that dimension has an extent of 0 and *allocate_object* is zero-sized.

A specified bound must not be an expression that contains as a primary an array inquiry function whose argument is an *allocate_object* in the same **ALLOCATE** statement. The *stat_variable* must not be allocated within the **ALLOCATE** statement in which it appears; nor can it depend on the value, bounds, allocation status, or association status of any *allocate_object* or subobject of an *allocate_object* allocated in the same statement.

If the **STAT=** specifier is not present and an error condition occurs during execution of the statement, the program terminates. If the **STAT=** specifier is present, the *stat_variable* is assigned one of the following values:

Stat value	Error condition
0	No error
1	Error in system routine attempting to do allocation
2	An invalid data object has been specified for allocation

Stat value	Error condition
3	Both error conditions 1 and 2 have occurred

Allocating an allocatable array that is already allocated causes an error condition in the **ALLOCATE** statement.

Pointer allocation creates an object that has the **TARGET** attribute. Additional pointers can be associated with this target (or a subobject of it) through pointer assignment. If you reallocate a pointer that is already associated with a target:
- A new target is created and the pointer becomes associated with this target
- Any previous association with the pointer is broken
- Any previous target that had been created by allocation and is not associated with any other pointers becomes inaccessible

Use the **ALLOCATED** intrinsic function to determine if an allocatable array is currently allocated. Use the **ASSOCIATED** intrinsic function to determine the association status of a pointer or whether a pointer is currently associated with a specified target.

Examples

```
CHARACTER, POINTER :: P(:,:)
CHARACTER, TARGET :: C(4,4)
INTEGER, ALLOCATABLE, DIMENSION(:) :: A
P => C
N = 2; M = N
ALLOCATE (P(N,M),STAT=I)        ! P is no longer associated with C
N = 3                           ! Target array for P maintains 2X2 shape
IF (.NOT.ALLOCATED(A)) ALLOCATE (A(N**2))
END
```

Related Information

- "ALLOCATABLE" on page 245
- "DEALLOCATE" on page 280
- "Allocation Status" on page 58
- "Pointer Association" on page 139
- "Deferred-Shape Arrays" on page 69
- "ALLOCATED(ARRAY)" on page 529
- "ASSOCIATED(POINTER, TARGET)" on page 533
- "Initialization" on page 6

ASSIGN

Purpose

The **ASSIGN** statement assigns a statement label to an integer variable.

Format

```
►►──ASSIGN──stmt_label──TO──variable_name──────────────────────────────►◄
```

stmt_label
: specifies the statement label of an executable statement or a **FORMAT** statement in the scoping unit containing the **ASSIGN** statement

variable_name
: is the name of a scalar **INTEGER(4)** or **INTEGER(8)** variable

Rules

A statement containing the designated statement label must appear in the same scoping unit as the **ASSIGN** statement.

- If the statement containing the statement label is an executable statement, you can use the label name in an assigned **GO TO** statement that is in the same scoping unit.
- If the statement containing the statement label is a **FORMAT** statement, you can use the label name as the format specifier in a **READ**, **WRITE**, or **PRINT** statement that is in the same scoping unit.

You can redefine an integer variable defined with a statement label value with the same or different statement label value or an integer value. However, you must define the variable with a statement label value before you reference it in an assigned **GO TO** statement or as a format identifier in an input/output statement.

The value of *variable_name* is not the integer constant represented by the label itself, and you cannot use it as such.

The **ASSIGN** statement has been deleted from Fortran 95.

Examples

```
      ASSIGN 30 TO LABEL
      NUM = 40
      GO TO LABEL
      NUM = 50                ! This statement is not executed
30    ASSIGN 1000 TO IFMT
      PRINT IFMT, NUM         ! IFMT is the format specifier
1000  FORMAT(1X,I4)
      END
```

Related Information

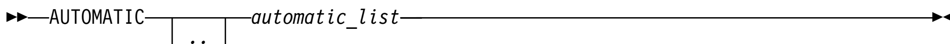
- “Statement Labels” on page 15
- “GO TO (Assigned)” on page 324
- “Deleted Features” on page 735

AUTOMATIC

Purpose

The **AUTOMATIC** attribute specifies that a variable has a storage class of automatic; that is, the variable is not defined once the procedure ends.

Format



automatic
: is a variable name or an array declarator with an explicit-shape specification list or a deferred-shape specification list

Rules

If *automatic* has the same name as the name of the function in which it is declared, it must not be of type character or of derived type.

Function results that are pointers or arrays, dummy arguments, statement functions, automatic objects, or pointees must not have the **AUTOMATIC** attribute. A variable with the **AUTOMATIC** attribute cannot be defined in the scoping unit of a module. A variable that is explicitly declared with the **AUTOMATIC** attribute cannot be a common block item.

A variable must not have the **AUTOMATIC** attribute specified more than once in the same scoping unit.

Any variable declared as **AUTOMATIC** within the scope of a thread's work will be local to that thread.

If the **-qinitauto** compiler option is not specified, a variable with the **AUTOMATIC** attribute cannot be initialized, either with a **DATA** statement or with a type declaration statement. If the **-qinitauto** option is specified, all bytes of storage for variables with the **AUTOMATIC** attribute are initialized to a specified byte value or, if no value is specified, to zero.

If *automatic* is a pointer, the **AUTOMATIC** attribute applies to the pointer itself, not to any target that is (or may become) associated with the pointer.

Local variables have a default storage class of automatic. See ″-qsave Option″ in the *User's Guide* for details on the default settings with regard to the invocation commands.

Note: An object with the **AUTOMATIC** attribute should not be confused with an automatic object. See "Automatic Objects" on page 29.

Attributes Compatible with the AUTOMATIC Attribute

- ALLOCATABLE
- DIMENSION
- POINTER
- TARGET
- VOLATILE

Examples

```
CALL SUB
CONTAINS
  SUBROUTINE SUB
    INTEGER, AUTOMATIC :: VAR
    VAR = 12
  END SUBROUTINE                  ! VAR becomes undefined
END
```

Related Information

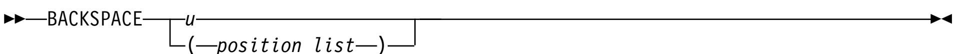
- "Storage Classes for Variables" on page 59
- "-qinitauto Option" in the *User's Guide*

BACKSPACE

Purpose

The **BACKSPACE** statement positions an external file, connected for sequential access, before the preceding record.

Format



u is an external unit identifier. The value of *u* must not be an asterisk or a Hollerith constant.

position_list
is a list that must contain one unit specifier ([**UNIT=**]*u*) and can also contain one of each of the other valid specifiers:

[UNIT=] *u*
is a unit specifier in which *u* must be an external unit identifier whose value is not an asterisk. An external unit identifier refers to an external file that is represented by a scalar integer expression, whose value is in the range 1 through 2147483647. If the optional characters **UNIT=** are omitted, *u* must be the first item in *position_list*.

IOSTAT= *ios*
is an input/output status specifier that specifies the status of the

input/output operation. *ios* is a scalar variable of type **INTEGER(4)** or default integer. When the **BACKSPACE** statement finishes executing, *ios* is defined with:
- A zero value if no error condition occurs
- A positive value if an error occurs.

ERR= *stmt_label*
is an error specifier that specifies the statement label of an executable statement in the same scoping unit to which control is to transfer in the case of an error. Coding the **ERR=** specifier suppresses error messages.

Rules

If there is no preceding record, the file position does not change. If the preceding record is the endfile record, the file is positioned before the endfile record. You cannot backspace over records that were written using list-directed or namelist formatting.

If the **ERR=** and **IOSTAT=** specifiers are set and an error is encountered, transfer is made to the statement specified by the **ERR=** specifier and a positive integer value is assigned to *ios*.

If **IOSTAT=** and **ERR=** are not specified,
- The program stops if a severe error is encountered.
- The program continues to the next statement if a recoverable error is encountered and the **ERR_RECOVERY** run-time option is set to **YES**. If the option is set to **NO**, the program stops.

Examples

```
   BACKSPACE 15
   BACKSPACE (UNIT=15,ERR=99)
   ⋮
99 PRINT *, "Unable to backspace file."
   END
```

Related Information

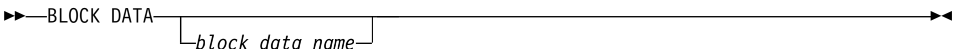
- “Conditions and IOSTAT Values” on page 193
- “Chapter 8. Input/Output Concepts” on page 183
- ″Setting Runtime Options for Input/Output″ in the *User's Guide*

BLOCK DATA

Purpose

A **BLOCK DATA** statement is the first statement in a block data program unit, which provides initial values for variables in named common blocks.

Format



block_data_name
: is the name of a block data program unit

Rules

You can have more than one block data program unit in an executable program, but only one can be unnamed.

The name of the block data program unit, if given, must not be the same as an external subprogram, entry, main program, module, or common block in the executable program. It also must not be the same as a local entity in this program unit.

Examples

```
BLOCK DATA ABC
  PARAMETER (I=10)
  DIMENSION Y(5)
  COMMON /L4/ Y
  DATA Y /5*I/
END BLOCK DATA ABC
```

Related Information

- "Block Data Program Unit" on page 156
- "END" on page 296 for details on the **END BLOCK DATA** statement

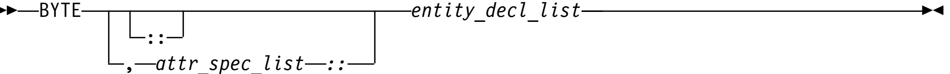
---

BYTE

Purpose

The **BYTE** type declaration statement specifies the attributes of objects and functions of type byte. Each scalar object has a length of 1. Initial values can be assigned to objects.

Format



where:

attr_spec
ALLOCATABLE **AUTOMATIC** **DIMENSION** (*array_spec*) **EXTERNAL** **INTENT** (*intent_spec*) **INTRINSIC** **OPTIONAL** **PARAMETER** **POINTER** **PRIVATE** **PUBLIC** **SAVE** **STATIC** **TARGET** **VOLATILE**

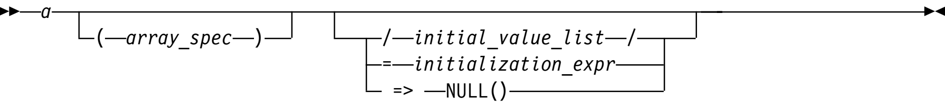
attr_spec
For detailed information on rules about a particular attribute, refer to the statement of the same name.

intent_spec
is either **IN**, **OUT**, or **INOUT**

:: is the double colon separator. It is required if attributes are specified, = *initialization_expr* or => **NULL()** is used.

array_spec
is a list of dimension bounds

entity_decl



a is an object name or function name. *array_spec* cannot be specified for a function name.

initial_value
provides an initial value for the entity specified by the immediately preceding name

initialization_expr
provides an initial value, by means of an initialization expression, for the entity specified by the immediately preceding name

=> **NULL()**
provides the initial value for the pointer object

Rules

Within the context of a derived type definition:

- If => appears in a component initialization, the **POINTER** attribute must appear in the *attr_spec_list*.
- If = appears in a component initialization, the **POINTER** attribute cannot appear in the component *attr_spec_list*.
- The compiler will evaluate *initialization_expr* within the scoping unit of the type definition.

If => appears for a variable, the object must have the **POINTER** attribute.

If *initialization_expr* appears for a variable, the object cannot have the **POINTER** attribute.

Entities in type declaration statements are constrained by the rules of any attributes specified for the entities, as detailed in the corresponding attribute statements.

The type declaration statement overrides the implicit type rules in effect. You can use a type declaration statement that confirms the type of an intrinsic function. The appearance of a generic or specific intrinsic function name in a type declaration statement does not cause the name to lose its intrinsic property.

An object cannot be initialized in a type declaration statement if it is a dummy argument, an allocatable array, a function result, an object in blank common, an integer pointer, an external name, an intrinsic name, or an automatic object. Nor can an object be initialized if it has the **AUTOMATIC** attribute. The object may be initialized if it appears in a named common block in a block data program unit or if it appears in a named common block in a module.

In Fortran 95, a pointer can be initialized. Pointers can only be initialized by the use of **=> NULL()**.

The specification expression of an *array_spec* can be a nonconstant expression if the specification expression appears in an interface body or in the specification part of a subprogram. Any object being declared that uses this nonconstant expression and is not a dummy argument or a pointee is called an *automatic object*.

An attribute cannot be repeated in a given type declaration statement, nor can an entity be explicitly given the same attribute more than once in a scoping unit.

initialization_expr must be specified if the statement contains the **PARAMETER** attribute. If the entity you are declaring is a variable, and *initalization_expr* or **NULL()** is specified, the variable is initially defined. If the entity you are declaring is a derived type component, and *initalization_expr* or **NULL()** is specified, the derived type has default initialization. *a* becomes defined with the value determined by *initialization_expr*, in accordance with the rules for intrinsic assignment. If the entity is an array, its shape must be specified either in the type declaration statement or in a previous specification statement in the same scoping unit. A variable or variable subobject cannot be initialized more than once. If *a* is a variable, the presence of *initialization_expr* or => **NULL()** implies that *a* is a saved object, except for an object in a named common block. The initialization of an object could affect the fundamental storage class of an object.

An *array_spec* specified in the *entity_decl* takes precedence over the *array_spec* in the **DIMENSION** attribute.

An array function result that does not have the **POINTER** attribute must have an explicit-shape array specification.

If the entity declared is a function, it must not have an accessible explicit interface unless it is an intrinsic function.

If `T` or `F`, defined previously as the name of a constant, appears in a type declaration statement, it is no longer an abbreviated logical constant but the name of the named constant.

Examples

```
BYTE, DIMENSION(4) :: X=(/1,2,3,4/)
```

Related Information

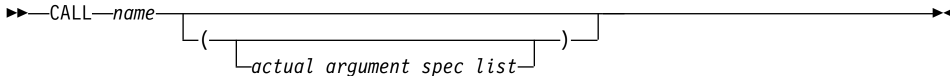
- "BYTE" on page 38
- "Initialization Expressions" on page 87
- "How Type Is Determined" on page 52, for details on the implicit typing rules
- "Automatic Objects" on page 29
- "Storage Classes for Variables" on page 59
- "DATA" on page 277, for details on initial values

CALL

Purpose

The **CALL** statement invokes a subroutine to be executed.

Format



name is the name of an internal, external, or module subroutine, an entry in an external or module subroutine, an intrinsic subroutine, or a generic name.

Rules

Executing a **CALL** statement results in the following order of events:

1. Actual arguments that are expressions are evaluated.
2. Actual arguments are associated with their corresponding dummy arguments.
3. Control transfers to the specified subroutine.
4. The subroutine is executed.
5. Control returns from the subroutine.

A subprogram can call itself recursively, directly or indirectly, if the subroutine statement specifies the **RECURSIVE** keyword.

An external subprogram can also refer to itself directly or indirectly if the **-qrecur** compiler option is specified.

If a **CALL** statement includes one or more alternate return specifiers among its arguments, control may be transferred to one of the statement labels indicated, depending on the action specified by the subroutine in the **RETURN** statement.

The argument list built-in functions **%VAL** and **%REF** are supplied to aid interlanguage calls by allowing arguments to be passed by value and by reference, respectively. They can only be specified in non-Fortran procedure references.

Examples

```
INTERFACE
  SUBROUTINE SUB3(D1,D2)
    REAL D1,D2
  END SUBROUTINE
END INTERFACE
ARG1=7 ; ARG2=8
```

```
CALL SUB3(D2=ARG2,D1=ARG1)    ! subroutine call with argument keywords
END

SUBROUTINE SUB3(F1,F2)
  REAL F1,F2,F3,F4
  F3 = F1/F2
  F4 = F1-F2
  PRINT *, F3, F4
END SUBROUTINE
```

Related Information

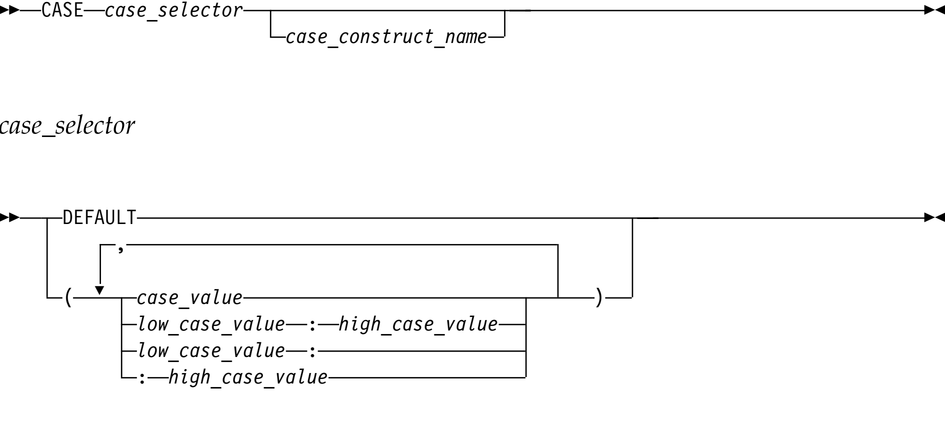
- "Recursion" on page 175
- "%VAL and %REF" on page 165
- "Actual Argument Specification" on page 161
- "Asterisks as Dummy Arguments" on page 172

CASE

Purpose

The **CASE** statement initiates a **CASE** statement block in a **CASE** construct, which has a concise syntax for selecting, at most, one of a number of statement blocks for execution.

Format



case_construct_name
: Is a name that identifies the **CASE** construct.

case_value
: is a scalar initialization expression of type integer, character, or logical

low_case_value, high_case_value
: are each scalar initialization expressions of type integer, character, or logical

Rules

The case index, determined by the **SELECT CASE** statement, is compared to each *case_selector* in a **CASE** statement. When a match occurs, the *stmt_block* associated with that **CASE** statement is executed. If no match occurs, no *stmt_block* is executed. No two case value ranges can overlap.

A match is determined as follows:

case_value

DATA TYPE: integer, character or logical
MATCH for integer and character: *case index* = *case_value*
MATCH for logical: *case index* **.EQV.** *case_value* is true

low_case_value : high_case_value

DATA TYPE: integer or character
MATCH: *low_case_value* ≤ *case index* ≤ *high_case_value*

low_case_value :

DATA TYPE: integer or character
MATCH: *low_case_value* ≤ *case index*

: high_case_value

DATA TYPE: integer or character
MATCH: *case index* ≤ *high_case_value*

DEFAULT

DATA TYPE: not applicable
MATCH: if no other match occurs.

There must be only one match. If there is a match, the statement block associated with the matched *case_selector* is executed, completing execution of the case construct. If there is no match, execution of the case construct is complete.

If the *case_construct_name* is specified, it must match the name specified on the **SELECT CASE** and **END SELECT** statements.

DEFAULT is the default *case_selector.* Only one of the **CASE** statements may have **DEFAULT** as the *case_selector*.

Each case value must be of the same data type as the *case_expr*, as defined in the **SELECT CASE** statement. If any typeless constants or **BYTE** named constants are encountered in the *case_selectors*, they are converted to the data type of the *case_expr*.

When the *case_expr* and the case values are of type character, they can have different lengths. If you specify the **-qctyplss** compiler option, a character constant expression used as the *case_expr* remains as type character. The character constant expression will not be treated as a typeless constant.

Examples

```
ZERO: SELECT CASE(N)

  CASE DEFAULT ZERO          ! Default CASE statement for
                             ! CASE construct ZERO
       OTHER: SELECT CASE(N)
          CASE(:-1)          ! CASE statement for CASE
                             ! construct OTHER
             SIGNUM = -1
          CASE(1:) OTHER
              SIGNUM = 1
       END SELECT OTHER
  CASE (0)
    SIGNUM = 0

END SELECT ZERO
```

Related Information

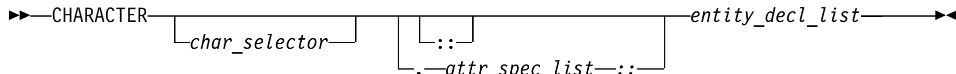
- “CASE Construct” on page 123
- “SELECT CASE” on page 390
- “END (Construct)” on page 298, for details on the **END SELECT** statement

CHARACTER

Purpose

A **CHARACTER** type declaration statement specifies the kind, length, and attributes of objects and functions of type character. Initial values can be assigned to objects.

Format

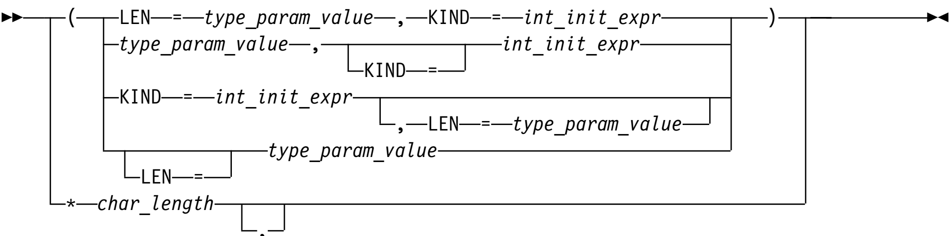


where:

attr_spec
ALLOCATABLE
AUTOMATIC
DIMENSION (*array_spec*)
EXTERNAL
INTENT (*intent_spec*)
INTRINSIC
OPTIONAL
PARAMETER
POINTER
PRIVATE
PUBLIC
SAVE
STATIC
TARGET
VOLATILE

char_selector

specifies the character length (number of characters between 0 and 256 MB). Values exceeding 256 MB are set to 256 MB in 32 bit, while negative values result in a length of zero. If not specified, the default length is 1. The kind type parameter, if specified, must be 1, which specifies the ASCII character representation.



type_param_value

is a specification expression or an asterisk (*)

int_init_expr

is a scalar integer initialization expression that must evaluate to 1

char_length

is either a scalar integer literal constant (which cannot specify a kind type parameter) or a *type_param_value* enclosed in parentheses

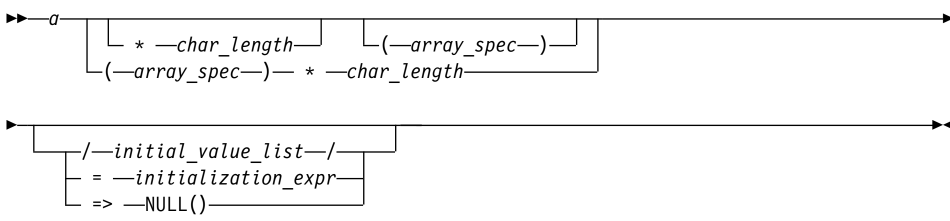
attr_spec
For detailed information on rules about a particular attribute, refer to the statement of the same name.

intent_spec
is either **IN**, **OUT**, or **INOUT**

::
is the double colon separator. It is required if attributes are specified,= *initialization_expr* or => **NULL()** is used

array_spec
is a list of dimension bounds

entity_decl



a
is an object name or function name. *array_spec* cannot be specified for a function name.

initial_value
provides an initial value for the entity specified by the immediately preceding name

initialization_expr
provides an initial value, by means of an initialization expression, for the entity specified by the immediately preceding name

=> **NULL()**
provides the initial value for the pointer object

Rules

Within the context of a derived type definition:

- If => appears in a component initialization, the **POINTER** attribute must appear in the *attr_spec_list*.
- If = appears in a component initialization, the **POINTER** attribute cannot appear in the component *attr_spec_list*.
- The compiler will evaluate *initialization_expr* within the scoping unit of the type definition.

If => appears for a variable, the object must have the **POINTER** attribute.

If *initialization_expr* appears for a variable, the object cannot have the **POINTER** attribute.

Entities in type declaration statements are constrained by the rules of any attributes specified for the entities, as detailed in the corresponding attribute statements.

The type declaration statement overrides the implicit type rules in effect. You can use a type declaration statement that confirms the type of an intrinsic function. The appearance of a generic or specific intrinsic function name in a type declaration statement does not cause the name to lose its intrinsic property.

An object cannot be initialized in a type declaration statement if it is a dummy argument, an allocatable array, a pointer, a function result, an object in blank common, an integer pointer, an external name, an intrinsic name, or an automatic object. Nor can an object be initialized if it has the **AUTOMATIC** attribute. The object may be initialized if it appears in a named common block in a block data program unit or if it appears in a named common block in a module.

In Fortran 95, a pointer can be initialized. Pointers can only be initialized by the use of **=> NULL()**.

The specification expression of a *type_param_value* or an *array_spec* can be a nonconstant expression if the specification expression appears in an interface body or in the specification part of a subprogram. Any object being declared that uses this nonconstant expression and is not a dummy argument or a pointee is called an *automatic object*.

An attribute cannot be repeated in a given type declaration statement, nor can an entity be explicitly given the same attribute more than once in a scoping unit.

initialization_expr must be specified if the statement contains the **PARAMETER** attribute. If the entity you are declaring is a variable, and *initialization_expr* or **NULL()** is specified, the variable is initially defined. If the entity you are declaring is a derived type component, and *initialization_expr* or **NULL()** is specified, the derived type has default initialization. *a* becomes defined with the value determined by *initialization_expr*, in accordance with the rules for intrinsic assignment. If the entity is an array, its shape must be specified either in the type declaration statement or in a previous specification statement in the same scoping unit. A variable or variable subobject cannot be initialized more than once. If *a* is a variable, the presence of *initialization_expr* or =>

NULL() implies that *a* is a saved object, except for an object in a named common block. The initialization of an object could affect the fundamental storage class of an object.

An *array_spec* specified in an *entity_decl* takes precedence over the *array_spec* in the **DIMENSION** attribute. A *char_length* specified in an *entity_decl* takes precedence over any length specified in *char_selector*.

An array function result that does not have the **POINTER** attribute must have an explicit-shape array specification.

If the entity declared is a function, it must not have an accessible explicit interface unless it is an intrinsic function.

If `T` or `F`, defined previously as the name of a constant, appears in a type declaration statement, it is no longer an abbreviated logical constant but the name of the named constant.

The optional comma after *char_length* in a **CHARACTER** type declaration statement is permitted only if no double colon separator (::) appears in the statement.

If the **CHARACTER** type declaration statement is in the scope of a module, block data program unit, or main program, and you specify the length of the entity as an inherited length, the entity must be the name of a named character constant. The character constant assumes the length of its corresponding expression defined by the **PARAMETER** attribute.

If the **CHARACTER** type declaration statement is in the scope of a procedure and the length of the entity is inherited, the entity name must be the name of a dummy argument or a named character constant. If the statement is in the scope of an external function, it can also be the function or entry name in a **FUNCTION** or **ENTRY** statement in the same program unit. If the entity name is the name of a dummy argument, the dummy argument assumes the length of the associated actual argument for each reference to the procedure. If the entity name is the name of a character constant, the character constant assumes the length of its corresponding expression defined by the **PARAMETER** attribute. If the entity name is a function or entry name, the entity assumes the length specified in the calling scoping unit.

The length of a character function is either a specification expression (which must be a constant expression if the function type is not declared in an interface block) or it is an asterisk, indicating the length of a dummy procedure name. The length cannot be an asterisk if the function is an internal or module function, if it is recursive, or if it returns array or pointer values.

Examples

```
I=7
CHARACTER(KIND=1,LEN=6) APPLES /'APPLES'/
CHARACTER(7), TARGET :: ORANGES = 'ORANGES'
CALL TEST(APPLES,I)
CONTAINS
  SUBROUTINE  TEST(VARBL,I)
    CHARACTER*(*), OPTIONAL :: VARBL   ! VARBL inherits a length of 6
    CHARACTER(I) :: RUNTIME            ! Automatic object with length of 7
  END SUBROUTINE
END
```

Related Information

- "Character" on page 35
- "Initialization Expressions" on page 87
- "How Type Is Determined" on page 52 for details on the implicit typing rules
- "Array Declarators" on page 65
- "Automatic Objects" on page 29
- "Storage Classes for Variables" on page 59
- "DATA" on page 277, for details on initial values

CLOSE

Purpose

The **CLOSE** statement disconnects an external file from a unit.

Format

```
►►—CLOSE—(—close_list—)——————————————————————————————————————►◄
```

close_list
: is a list that must contain one unit specifier (**UNIT=***u*) and can also contain one of each of the other valid specifiers. The valid specifiers are:

[UNIT=] *u*
: is a unit specifier in which *u* must be an external unit identifier whose value is not an asterisk. An external unit identifier refers to an external file that is represented by a scalar integer expression, whose value is in the range 1 through 2147483647. If the optional characters **UNIT=** are omitted, *u* must be the first item in *close_list*.

IOSTAT= *ios*
: is an input/output status specifier that specifies the status of the input/output operation. *ios* is a scalar variable of type **INTEGER(4)** or default integer. When the input/output statement containing this specifier finishes executing, *ios* is defined with:

- A zero value if no error condition occurs
- A positive value if an error occurs.

ERR= *stmt_label*
is an error specifier that specifies the statement label of an executable statement in the same scoping unit to which control is to transfer in the case of an error. Coding the **ERR=** specifier suppresses error messages.

STATUS= *char_expr*
specifies the status of the file after it is closed. *char_expr* is a scalar character expression whose value, when any trailing blanks are removed, is either **KEEP** or **DELETE**.
- If **KEEP** is specified for a file that exists, the file will continue to exist after the **CLOSE** statement. If **KEEP** is specified for a file that does not exist, the file will not exist after the **CLOSE** statement. **KEEP** must not be specified for a file whose status prior to executing the **CLOSE** statement is **SCRATCH**.
- If **DELETE** is specified, the file will not exist after the **CLOSE** statement.

The default is **DELETE** if the file status is **SCRATCH**; otherwise, the default is **KEEP**.

Rules

A **CLOSE** statement that refers to a unit can occur in any program unit of an executable program and need not occur in the same scoping unit as the **OPEN** statement referring to that unit. You can specify a unit that does not exist or has no file connected; the **CLOSE** statement has no effect in this case.

Unit 0 cannot be closed.

When an executable program stops for reasons other than an error condition, all units that are connected are closed. Each unit is closed with the status **KEEP** unless the file status prior to completion was **SCRATCH**, in which case the unit is closed with the status **DELETE**. The effect is as though a **CLOSE** statement without a **STATUS=** specifier were executed on each connected unit.

If a preconnected unit is disconnected by a **CLOSE** statement, the rules of implicit opening apply if the unit is later specified in a **WRITE** statement (without having been explicitly opened).

Examples

```
CLOSE(15)
CLOSE(UNIT=16,STATUS='DELETE')
```

Related Information

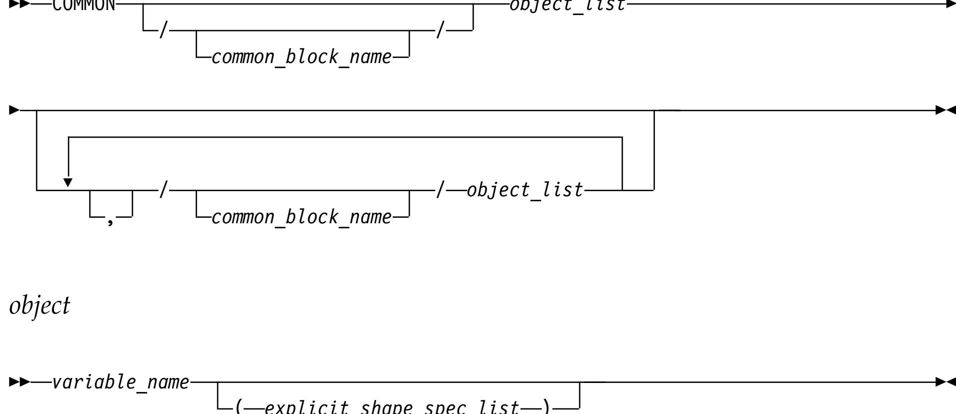
- “Connection of a Unit” on page 186
- “Conditions and IOSTAT Values” on page 193
- “OPEN” on page 355

COMMON

Purpose

The **COMMON** statement specifies common blocks and their contents. A common block is a storage area that two or more scoping units can share, allowing them to define and reference the same data and to share storage units.

Format



Rules

object cannot refer to a dummy argument, automatic object, allocatable array, pointee, function, function result, or entry to a procedure. *object* cannot have the **STATIC** or **AUTOMATIC** attributes.

If an *explicit_shape_spec_list* is present, *variable_name* must not have the **POINTER** attribute. Each dimension bound must be a constant specification expression. This form specifies that *variable_name* has the **DIMENSION** attribute.

If *object* is of derived type, it must be a sequence derived type. Given a sequenced structure where all the ultimate components are nonpointers, and are all of character type or all of type default integer, default real, default complex, default logical or double precision real, the structure is treated as if its components are enumerated directly in the common block.

A pointer object in a common block can only be storage associated with pointers of the same type, type parameters, and rank.

An object in a common block with **TARGET** attribute can be storage associated with another object. That object must have the **TARGET** attribute and have the same type and type parameters.

Pointers of type **BYTE** can be storage associated with pointers of type **INTEGER(1)** and **LOGICAL(1)**. Integer and logical pointers of the same length can be storage associated if you specify the **-qintlog** compiler option.

If you specify *common_block_name*, all variables specified in the *object_list* that follows are declared to be in that named common block. If you omit *common_block_name*, all variables that you specify in the *object_list* that follows are in the blank common block.

Within a scoping unit, a common block name can appear more than once in the same or in different **COMMON** statements. Each successive appearance of the same common block name continues the common block specified by that name. Common block names are global entities.

The variables in a common block can have different data types. You can mix character and noncharacter data types within the same common block. Variable names in common blocks can appear in only one **COMMON** statement in a scoping unit, and you cannot duplicate them within the same **COMMON** statement.

By default, common blocks are shared across threads, and so the use of the **COMMON** statement is thread-unsafe if any storage unit in the common block needs to be updated by more than one thread, or is updated by one thread and referenced by another. To ensure your application uses **COMMON** in a thread-safe manner, you must either serialize access to the data using locks, or make certain that the common blocks are local to each thread. The **Pthreads** library module provides mutexes to allow you to serialize access to the data using locks. See “Chapter 15. Pthreads Library Module” on page 677 for more information. The *lock_name* attribute on the **CRITICAL** directive also provides the ability to serialize access to data. See “CRITICAL / END CRITICAL” on page 453 for more information. The **THREADLOCAL** directive ensures that common blocks are local to each thread. See “THREADLOCAL” on page 507 and “THREADPRIVATE” on page 510 for more information.

Common Association

Within an executable program, all nonzero-sized named common blocks with the same name have the same first storage unit. There can be one blank common block, and all scoping units that refer to nonzero-sized blank common refer to the same first storage unit.

All zero-sized common blocks with the same name are storage-associated with one another. All zero-sized blank common blocks are associated with one another and with the first storage unit of any nonzero-sized blank common blocks. Use association or host association can cause these associated objects to be accessible in the same scoping unit.

Because association is by storage unit, variables in a common block can have different names and types in different scoping units.

Common Block Storage Sequence: Storage units for variables within a common block in a scoping unit are assigned in the order that their names appear within the **COMMON** statement.

You can extend a common block by using an **EQUIVALENCE** statement, but only by adding beyond the last entry, not before the first entry. For example, these statements specify X:

```
COMMON /X/ A,B      ! common block named X
REAL C(2)
EQUIVALENCE (B,C)
```

The contents of common block X are as follows:

```
             |    |    |    |    |    |    |    |    |    |    |    |    |
Variable A:  |         A         |
Variable B:                      |         B         |
Array C:                         |        C(1)       |        C(2)       |
```

Only **COMMON** and **EQUIVALENCE** statements that appear in a scoping unit contribute to the common block storage sequences formed in that unit, not including variables in common made accessible by use association or host association.

An **EQUIVALENCE** statement cannot cause the storage sequences of two different common blocks to become associated. While a common block can be declared in the scoping unit of a module, it must not be declared in another scoping unit that accesses entities from the module through use association.

Use of **COMMON** can lead to misaligned data. Any use of misaligned data can adversely affect the performance of the program.

Size of a Common Block: The size of a common block is equal to the number of bytes of storage needed to hold all the variables in the common block, including any extensions resulting from equivalence association.

Differences Between Named and Blank Common Blocks:
- Within an executable program, there can be more than one named common block, but only one blank common block.

- In all scoping units of an executable program, named common blocks of the same name must have the same size, but blank common blocks can have different sizes. (If you specify blank common blocks with different sizes in different scoping units, the length of the longest block becomes the length of the blank common block in the executable program.)
- You can initially define objects in a named common block by using a **BLOCK DATA** program unit containing a **DATA** statement or a type declaration statement. You cannot initially define any elements of a common block in a blank common block.

  If a named common block, or any part of it, is initialized in more than one scoping unit, the initial value is undefined. To avoid this problem, use block data program units or modules to initialize named common blocks; each named common block should be initialized in only one block data program unit or module.

Examples

```
INTEGER MONTH,DAY,YEAR
COMMON /DATE/ MONTH,DAY,YEAR
REAL          R4
REAL          R8
CHARACTER(1)  C1
COMMON /NOALIGN/ R8,C1,R4     ! R4 will not be aligned on a
                              ! full-word boundary
```

Related Information

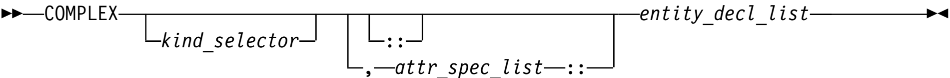
- “Chapter 15. Pthreads Library Module” on page 677
- “THREADLOCAL” on page 507
- “Block Data Program Unit” on page 156
- “Explicit-Shape Arrays” on page 66
- “The Scope of a Name” on page 134, for details on global entities
- “Storage Classes for Variables” on page 59

COMPLEX

Purpose

A **COMPLEX** type declaration statement specifies the length and attributes of objects and functions of type complex. Initial values can be assigned to objects.

Format



where:

attr_spec
ALLOCATABLE **AUTOMATIC** **DIMENSION** (*array_spec*) **EXTERNAL** **INTENT** (*intent_spec*) **INTRINSIC** **OPTIONAL** **PARAMETER** **POINTER** **PRIVATE** **PUBLIC** **SAVE** **STATIC** **TARGET** **VOLATILE**

kind_selector

```
►►──┬─(──┬──────────┬──int_initialization_expr──)──┬──────────────────────►◄
    │    └─KIND── = ─┘                               │
    └─ * ──int_literal_constant──────────────────────┘
```

specifies the length of complex entities:
- If *int_initialization_expr* is specified, the valid values are 4, 8 and 16. These values represent the precision and range of each part of the complex entity.
- If the **int_literal_constant* form is specified, the valid values are 8, 16 and 32. These values represent the length of the whole complex entity, and correspond to the values allowed for the alternative form. *int_literal_constant* cannot specify a kind type parameter.

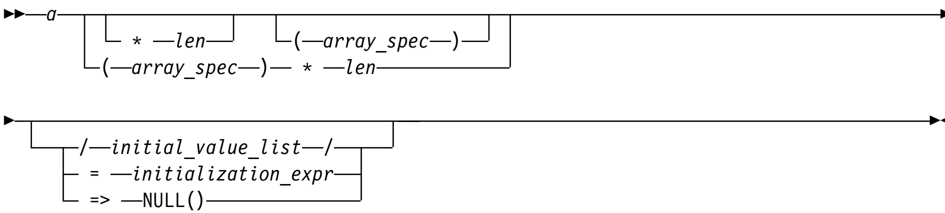
attr_spec
: For detailed information on rules about a particular attribute, refer to the statement of the same name.

intent_spec
: is either **IN**, **OUT**, or **INOUT**

::
: is the double colon separator. It is required if attributes are specified, = *initialization_expr* or => **NULL()** is used.

array_spec
: is a list of dimension bounds

entity_decl



a
: is an object name or function name. *array_spec* cannot be specified for a function name.

len
: overrides the length as specified in *kind_selector*, and cannot specify a kind type parameter. The entity length must be an integer literal constant that represents one of the permissible length specifications.

initial_value
: provides an initial value for the entity specified by the immediately preceding name

initialization_expr
: provides an initial value, by means of an initialization expression, for the entity specified by the immediately preceding name

=> **NULL()**
: provides an initial value for the pointer object

Rules

Within the context of a derived type definition:

- If => appears in a component initialization, the **POINTER** attribute must appear in the *attr_spec_list*.
- If = appears in a component initialization, the **POINTER** attribute cannot appear in the component *attr_spec_list*.
- The compiler will evaluate *initialization_expr* within the scoping unit of the type definition.

If => appears for a variable, the object must have the **POINTER** attribute.

If *initialization_expr* appears for a variable, the object cannot have the **POINTER** attribute.

Entities in type declaration statements are constrained by the rules of any attributes specified for the entities, as detailed in the corresponding attribute statements.

The type declaration statement overrides the implicit type rules in effect. You can use a type declaration statement that confirms the type of an intrinsic function. The appearance of a generic or specific intrinsic function name in a type declaration statement does not cause the name to lose its intrinsic property.

An object cannot be initialized in a type declaration statement if it is a dummy argument, an allocatable array, a pointer, a function result, an object in blank common, an integer pointer, an external name, an intrinsic name, or an automatic object. Nor can an object be initialized if it has the **AUTOMATIC** attribute. The object may be initialized if it appears in a named common block in a block data program unit or if it appears in a named common block in a module.

In Fortran 95, a pointer can be initialized. Pointers can only be initialized by the use of **=> NULL()**.

The specification expression of an *array_spec* can be a nonconstant expression if the specification expression appears in an interface body or in the specification part of a subprogram. Any object being declared that uses this nonconstant expression and is not a dummy argument or a pointee is called an *automatic object*.

An attribute cannot be repeated in a given type declaration statement, nor can an entity be explicitly given the same attribute more than once in a scoping unit.

initialization_expr must be specified if the statement contains the **PARAMETER** attribute. If the entity you are declaring is a variable, and *initialization_expr* or **NULL()** is specified, the variable is initially defined. If the entity you are declaring is a derived type component, and *initialization_expr* or **NULL()** is specified, the derived type has default initialization. *a* becomes defined with the value determined by *initialization_expr*, in accordance with the rules for intrinsic assignment. If the entity is an array, its shape must be specified either in the type declaration statement or in a previous specification statement in the same scoping unit. A variable or variable subobject cannot be initialized more than once. If *a* is a variable, the presence of *initialization_expr* or => **NULL()** implies that *a* is a saved object, except for an object in a named common block. The initialization of an object could affect the fundamental storage class of an object.

An *array_spec* specified in the *entity_decl* takes precedence over the *array_spec* in the **DIMENSION** attribute.

An array function result that does not have the **POINTER** attribute must have an explicit-shape array specification.

If the entity declared is a function, it must not have an accessible explicit interface unless it is an intrinsic function.

If T or F, defined previously as the name of a constant, appears in a type declaration statement, it is no longer an abbreviated logical constant but the name of the named constant.

Examples

```
COMPLEX, DIMENSION (2,3) :: ABC(3) ! ABC has 3 (not 6) array elements
```

Related Information

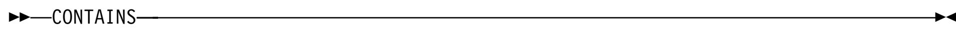
- "Complex" on page 33
- "Initialization Expressions" on page 87
- "How Type Is Determined" on page 52, for details on the implicit typing rules
- "Array Declarators" on page 65
- "Automatic Objects" on page 29
- "Storage Classes for Variables" on page 59
- "DATA" on page 277, for details on initial values

CONTAINS

Purpose

The **CONTAINS** statement separates the body of a main program, external subprogram, or module subprogram from any internal subprograms that it may contain. Similarly, it separates the specification part of a module from any module subprograms.

Format



Rules

When a **CONTAINS** statement exists, at least one subprogram must follow it.

The **CONTAINS** statement cannot appear in a block data program unit or in an internal subprogram.

Any label of a **CONTAINS** statement is considered part of the main program, subprogram, or module that contains the **CONTAINS** statement.

Examples

```
MODULE A
   ⋮
  CONTAINS                ! Module subprogram must follow
  SUBROUTINE B(X)
   ⋮
    CONTAINS              ! Internal subprogram must follow
    FUNCTION C(Y)
   ⋮
    END FUNCTION
  END SUBROUTINE
END MODULE
```

Related Information

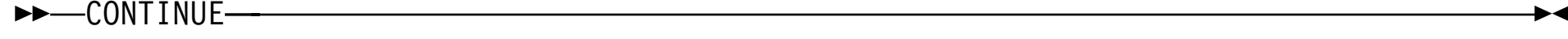
"Program Units, Procedures, and Subprograms" on page 141

CONTINUE

Purpose

The **CONTINUE** statement is an executable control statement that takes no action; it has no effect. This statement is often used as the terminal statement of a loop.

Format



Examples

```
     DO 100 I = 1,N
        X = X + N
100  CONTINUE
```

Related Information

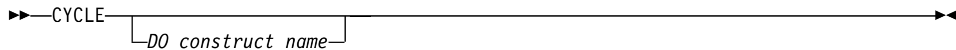
"Chapter 6. Control" on page 121

CYCLE

Purpose

The **CYCLE** statement terminates the current execution cycle of a **DO** or **DO WHILE** construct.

Format



DO_construct_name
is the name of a **DO** or **DO WHILE** construct

Rules

The **CYCLE** statement is placed within a **DO** or **DO WHILE** construct and belongs to the particular **DO** or **DO WHILE** construct specified by *DO_construct_name* or, if not specified, to the **DO** or **DO WHILE** construct that immediately surrounds it. The statement terminates only the current cycle of the construct that it belongs to.

When the **CYCLE** statement is executed, the current execution cycle of the **DO** or **DO WHILE** construct is terminated. Any executable statements after the **CYCLE** statement, including any terminating labeled action statement, will not be executed. For **DO** constructs, program execution continues with incrementation processing, if any. For **DO WHILE** constructs, program execution continues with loop control processing.

A **CYCLE** statement can have a statement label. However, it cannot be used as a labeled action statement that terminates a **DO** construct.

Examples

```
LOOP1: DO I = 1, 20
   N = N + 1
   IF (N > NMAX) CYCLE LOOP1          ! cycle to LOOP1

   LOOP2: DO WHILE (K==1)
      IF (K > KMAX) CYCLE             ! cycle to LOOP2
      K = K + 1
   END DO LOOP2

   LOOP3:  DO J = 1, 10
      N = N + 1
      IF (N > NMAX) CYCLE LOOP1       ! cycle to LOOP1
      CYCLE LOOP3                     ! cycle to LOOP3
   END DO LOOP3

END DO LOOP1
END
```

Related Information

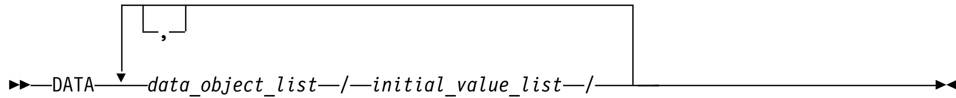
- "DO" on page 284
- "DO WHILE" on page 286

DATA

Purpose

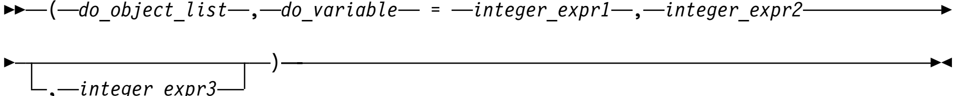
The **DATA** statement provides initial values for variables.

Format



data_object
: is a variable or an implied-**DO** list. Any subscript or substring expression must be an initialization expression.

*implied-***DO** *list*

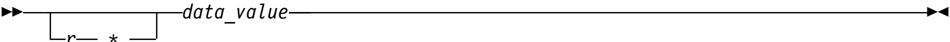


do_object
: is an array element, scalar structure component, substring, or implied-**DO** list

do_variable
: is a named scalar integer variable called the implied-**DO** variable. This variable is a statement entity.

integer_expr1, *integer_expr2*, and *integer_expr3*
: are each scalar integer expressions. The primaries of an expression can only contain constants or implied-**DO** variables of other implied-**DO** lists that have this implied-**DO** list within their ranges. Each operation must be intrinsic.

initial_value



r
: is a nonnegative scalar integer constant, or a nonnegative scalar integer subobject of a constant. If *r* is a named constant, or a subobject of a named constant, it must have been declared previously in the scoping unit or made accessible by use or host association. If *r* is a subobject of a constant, any subscript in it is an initialization expression. If *r* is omitted, the default value is 1. The form *r*data_value* is equivalent to *r* successive appearances of the data value.

data_value
: is a scalar constant, scalar subobject of a constant, signed integer literal constant, signed real literal constant, structure constructor, or **NULL()**

Rules

Specifying a non-pointer array object as a *data_object* is the same as specifying a list of all the elements in the array object in the order they are stored. An array with pointer attribute has only one corresponding initial value which is **NULL()**. Each *data_object_list* must specify the same number of items as its corresponding *initial_value_list*. There is a one-to-one correspondence between the items in these two lists. This correspondence establishes the initial value of each *data_object*.

For pointer initialization, if the *data_value* is **NULL()** then the corresponding *data_object* must have pointer attribute. If the *data_object* has pointer attribute then the corresponding *data_value* must be **NULL()**.

The definition of each *data_object* by its corresponding *initial_value* must follow the rules for intrinsic assignment, except as noted under "Using Typeless Constants" on page 49.

If *initial_value* is a structure constructor, each component must be an initialization expression. If *data_object* is a variable, any substring, subscript, or stride expressions must be initialization expressions.

If *data_value* is a named constant or structure constructor, the named constant or derived type must have been declared previously in the scoping unit or made accessible by use or host association.

Zero-sized arrays, implied-**DO** lists with iteration counts of zero, and values with a repeat factor of zero contribute no variables to the expanded *initial_value_list*, although a zero-length scalar character variable contributes one variable to the list.

You can use an implied-**DO** list in a **DATA** statement to initialize array elements, scalar structure components and substrings. The implied-**DO** list is expanded into a sequence of scalar structure components, array elements, or

substrings, under the control of the implied-**DO** variable. Array elements and scalar structure components must not have constant parents. Each scalar structure component must contain at least one component reference that specifies a subscript list.

The range of an implied-**DO** list is the *do_object_list*. The iteration count and the values of the implied-**DO** variable are established from *integer_expr1*, *integer_expr2*, and *integer_expr3*, the same as for a **DO** statement. When the implied-**DO** list is executed, it specifies the items in the *do_object_list* once for each iteration of the implied-**DO** list, with the appropriate substitution of values for any occurrence of the implied-**DO** variables. If the implied-**DO** variable has an iteration count of 0, no variables are added to the expanded sequence.

Each subscript expression in a *do_object* can only contain constants or implied-**DO** variables of implied-**DO** lists that have the subscript expression within their ranges. Each operation must be intrinsic.

To initialize list items of type logical with logical constants, you can also use the abbreviated forms (`T` for `.TRUE.` and `F` for `.FALSE.`). If `T` or `F` is a constant name that was defined previously with the **PARAMETER** attribute, XL Fortran recognizes the string as the named constant and assigns its value to the corresponding list item in the **DATA** statement.

In a block data program unit, you can use a **DATA** statement or type declaration statement to provide an initial value for a variable in a named common block.

In an internal or module subprogram, if the *data_object* is the same name as an entity in the host, and the *data_object* is not declared in any other specification statement in the internal subprogram, the *data_object* must not be referenced or defined before the **DATA** statement.

A **DATA** statement cannot provide an initial value for:
- An automatic object
- A dummy argument
- A pointee
- A variable in a blank common block
- The result variable of a function
- A data object whose storage class is automatic
- A variable that has the **ALLOCATABLE** attribute

You must not initialize a variable more than once in an executable program. If you associate two or more variables, you can only initialize one of the data objects.

Examples

Example 1:

```
      INTEGER Z(100),EVEN_ODD(0:9)
      LOGICAL FIRST_TIME
      CHARACTER*10 CHARARR(1)
      DATA    FIRST_TIME / .TRUE. /
      DATA    Z  / 100* 0 /
! Implied-DO list
      DATA  (EVEN_ODD(J),J=0,8,2) / 5 * 0 /  &
     &      ,(EVEN_ODD(J),J=1,9,2) / 5 * 1 /
! Nested example
      DIMENSION TDARR(3,4)  ! Initializes a two-dimensional array
      DATA ((TDARR(I,J),J=1,4),I=1,3) /12 * 0/
! Character substring example
      DATA (CHARARR(J)(1:3),J=1,1) /'aaa'/
      DATA (CHARARR(J)(4:7),J=1,1) /'bbbb'/
      DATA (CHARARR(J)(8:10),J=1,1) /'ccc'/
! CHARARR(1) contains 'aaabbbbccc'
```

Example 2:

```
TYPE DT
      INTEGER :: COUNT(2)
END TYPE DT

TYPE(DT), PARAMETER, DIMENSION(3) :: SPARM = DT ( (/3,5/) )

INTEGER :: A(5)

DATA A /SPARM(2)%COUNT(2) * 10/
```

Related Information

- “Chapter 3. Data Types and Data Objects” on page 27
- “Executing a DO Statement” on page 127
- “Statement and Construct Entities” on page 136

DEALLOCATE

Purpose

The **DEALLOCATE** statement dynamically deallocates allocatable arrays and pointer targets. A specified pointer becomes disassociated, while any other pointers associated with the target become undefined.

Format

```
>>—DEALLOCATE—(—object_list——————————————————————————)——————————————><
                           └—,—STAT— = —stat_variable—┘
```

object is a pointer or an allocatable array

stat_variable
is a scalar integer variable

Rules

An allocatable array that appears in a **DEALLOCATE** statement must be currently allocated. An allocatable array with the **TARGET** attribute cannot be deallocated through an associated pointer. Deallocation of such an array causes the association status of any associated pointer to become undefined. An allocatable array that has an undefined allocation status cannot be subsequently referenced, defined, allocated, or deallocated. Successful execution of a **DEALLOCATE** statement causes the allocation status of an allocatable array to become not allocated.

A pointer that appears in a **DEALLOCATE** statement must be associated with a whole target that was created with an **ALLOCATE** statement. Deallocation of a pointer target causes the association status of any other pointer associated with all or part of the target to become undefined.

Tips

Use the **DEALLOCATE** statement instead of the **NULLIFY** statement if no other pointer is associated with the allocated memory.

Deallocate memory that a pointer function has allocated.

If the **STAT=** specifier is not present and an error condition occurs during execution of the statement, the program terminates. If the **STAT=** specifier is present, *stat_variable* is assigned one of the following values:

Stat value	Error condition
0	No error
1	Error in system routine attempting to do deallocation
2	An invalid data object has been specified for deallocation
3	Both error conditions 1 and 2 have occurred

The *stat_variable* must not be deallocated within the **DEALLOCATE** statement in which it appears.

Examples

```
INTEGER, ALLOCATABLE :: A(:,:)
INTEGER X,Y

   ⋮

ALLOCATE (A(X,Y))
```

```
    ⋮

DEALLOCATE (A,STAT=I)
END
```

Related Information

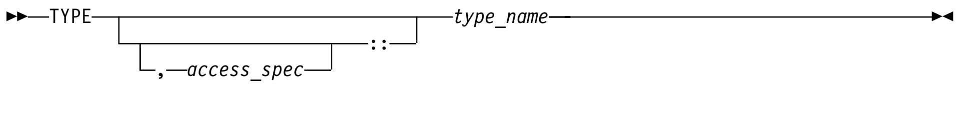
- "ALLOCATE" on page 246
- "ALLOCATABLE" on page 245
- "Allocation Status" on page 58
- "Pointer Association" on page 139
- "Deferred-Shape Arrays" on page 69

Derived Type

Purpose

The **Derived Type** statement is the first statement of a derived-type definition.

Format



access_spec
: is either **PRIVATE** or **PUBLIC**

type_name
: is the name of the derived type

Rules

access_spec can only be specified if the derived-type definition is within the specification part of a module.

type_name cannot be the same as the name of any intrinsic type, except **BYTE**, or the name of any other accessible derived type.

If a label is specified on the **Derived Type** statement, the label belongs to the scoping unit of the derived-type definition.

If the corresponding **END TYPE** statement specifies a name, it must be the same as *type_name*.

Examples

```
MODULE ABC
  TYPE, PRIVATE :: SYSTEM      ! Derived type SYSTEM can only be accessed
    SEQUENCE                   !   within module ABC
    REAL :: PRIMARY
```

```
    REAL :: SECONDARY
    CHARACTER(20), DIMENSION(5) :: STAFF
  END TYPE
END MODULE
```

Related Information

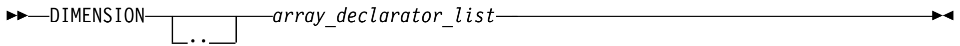
- "Derived Types" on page 39
- "END TYPE" on page 301
- "SEQUENCE" on page 391
- "PRIVATE" on page 370

DIMENSION

Purpose

The **DIMENSION** attribute specifies the name and dimensions of an array.

Format



Rules

According to Fortran 95, you can specify an array with up to seven dimensions.

With XL Fortran, you can specify up to 20 dimensions.

Only one dimension specification for an array name can appear in a scoping unit.

Attributes Compatible with the DIMENSION Attribute

- ALLOCATABLE
- AUTOMATIC
- INTENT
- OPTIONAL
- PARAMETER
- POINTER
- PRIVATE
- PUBLIC
- SAVE
- STATIC
- TARGET
- VOLATILE

Examples

```
CALL SUB(5,6)
CONTAINS
SUBROUTINE SUB(I,M)
  DIMENSION LIST1(I,M)                        ! automatic array
  INTEGER, ALLOCATABLE, DIMENSION(:,:) :: A   ! deferred-shape array

   :
```

```
.

END SUBROUTINE
END
```

Related Information

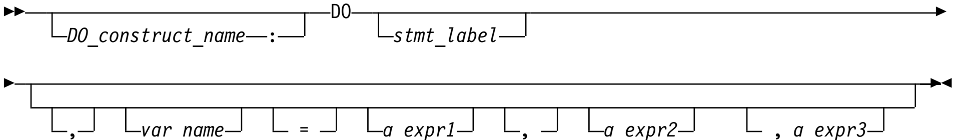
- “Chapter 4. Array Concepts” on page 63
- “VIRTUAL” on page 410

DO

Purpose

The **DO** statement controls the execution of the statements that follow it, up to and including a specified terminal statement. Together, these statements form a **DO** construct.

Format



DO_construct_name
: is a name that identifies the **DO** construct.

stmt_label
: is the statement label of an executable statement appearing after the **DO** statement in the same scoping unit. This statement denotes the end of the **DO** construct.

var_name
: is a scalar variable name of type integer or real, called the **DO** variable

*a_expr1***,** *a_expr2***, and** *a_expr3*
: are each scalar expressions of type integer or real

Rules

If you specify a *DO_construct_name* on the **DO** statement, you must terminate the construct with an **END DO** and the same *DO_construct_name*. Conversely, if you do not specify a *DO_construct_name* on the **DO** statement, and you terminate the **DO** construct with an **END DO** statement, you must not have a *DO_construct_name* on the **END DO** statement.

If you specify a statement label in the **DO** statement, you must terminate the **DO** construct with a statement that is labeled with that statement label. You can terminate a labeled **DO** statement with an **END DO** statement that is

labeled with that statement label, but you cannot terminate it with an unlabeled **END DO** statement. If you do not specify a label in the **DO** statement, you must terminate the **DO** construct with an **END DO** statement.

If the control clause (the clause beginning with *var_name*) is absent, the statement is an infinite **DO**. The loop will iterate indefinitely until interrupted (for example, by the **EXIT** statement).

When compiling a **DO** loop using the XL Fortran compiler, you should consider whether inserting an **INDEPENDENT** directive immediately preceding each loop is valid. If the iterations of the **DO** loop cannot be executed in an arbitrary order, the **INDEPENDENT** directive is not valid. The directive specifies that each iteration in the **DO** loop can be executed in any order without affecting the semantics of the program. For more information on the **INDEPENDENT** directive, see "INDEPENDENT" on page 466.

Examples

```
INTEGER :: SUM=0
OUTER: DO
  INNER: DO M=1,10
    READ (5,*) J
    IF (J.LE.I) THEN
      PRINT *, 'VALUE MUST BE GREATER THAN ', I
      CYCLE INNER
    END IF
    SUM=SUM+J
    IF (SUM.GT.500) EXIT OUTER
    IF (SUM.GT.100) EXIT INNER
  END DO INNER
  SUM=SUM+I
  I=I+10
END DO OUTER
PRINT *, 'SUM =',SUM
END
```

Related Information

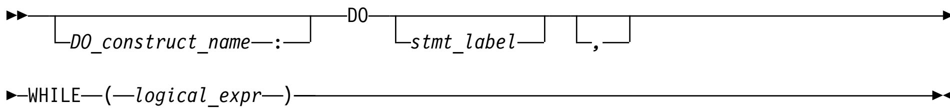
- "DO Construct" on page 126
- "END (Construct)" on page 298, for details on the **END DO** statement
- "EXIT" on page 308
- "CYCLE" on page 276
- "INDEPENDENT" on page 466
- "ASSERT" on page 445
- "CNCALL" on page 452
- "PERMUTATION" on page 487
- "PARALLEL DO / END PARALLEL DO" on page 479

DO WHILE

Purpose

The **DO WHILE** statement is the first statement in the **DO WHILE** construct, which indicates that you want the following statement block, up to and including a specified terminal statement, to be repeatedly executed for as long as the logical expression specified in the statement continues to be true.

Format



DO_construct_name
: is a name that identifies the **DO WHILE** construct

stmt_label
: is the statement label of an executable statement appearing after the **DO WHILE** statement in the same scoping unit. It denotes the end of the **DO WHILE** construct.

logical_expr
: is a scalar logical expression

Rules

If you specify a *DO_construct_name* on the **DO WHILE** statement, you must terminate the construct with an **END DO** and the same *DO_construct_name*. Conversely, if you do not specify a *DO_construct_name* on the **DO WHILE** statement, and you terminate the **DO WHILE** construct with an **END DO** statement, you must not have a *DO_construct_name* on the **END DO** statement.

If you specify a statement label in the **DO WHILE** statement, you must terminate the **DO WHILE** construct with a statement that is labeled with that statement label. You can terminate a labeled **DO WHILE** statement with an **END DO** statement that is labeled with that statement label, but you cannot terminate it with an unlabeled **END DO** statement. If you do not specify a label in the **DO WHILE** statement, you must terminate the **DO WHILE** construct with an **END DO** statement.

Examples

```
     MYDO: DO 10 WHILE (I .LE. 5)  ! MYDO is the construct name
        SUM = SUM + INC
        I = I + 1
10   END DO MYDO
     END
```

```
      SUBROUTINE EXAMPLE2
        REAL X(10)
        LOGICAL FLAG1
        DATA    FLAG1 /.TRUE./
        DO 20 WHILE (I .LE. 10)
           X(I) = A
           I = I + 1
20      IF (.NOT. FLAG1) STOP
      END SUBROUTINE EXAMPLE2
```

Related Information

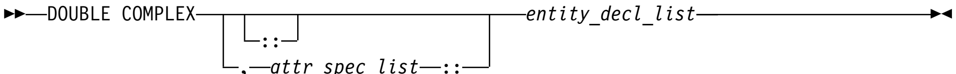
- "DO WHILE Construct" on page 130
- "END (Construct)" on page 298, for details on the **END DO** statement
- "EXIT" on page 308
- "CYCLE" on page 276

DOUBLE COMPLEX

Purpose

A **DOUBLE COMPLEX** type declaration statement specifies the attributes of objects and functions of type double complex. Initial values can be assigned to objects.

Format



where:

attr_spec
ALLOCATABLE
AUTOMATIC
DIMENSION (*array_spec*)
EXTERNAL
INTENT (*intent_spec*)
INTRINSIC
OPTIONAL
PARAMETER
POINTER
PRIVATE
PUBLIC
SAVE
STATIC
TARGET
VOLATILE

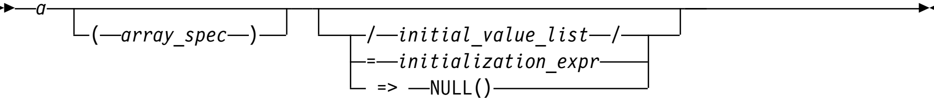
attr_spec
: For detailed information on rules about a particular attribute, refer to the statement of the same name.

intent_spec
: is either **IN**, **OUT**, or **INOUT**

::
: is the double colon separator. It is required if attributes are specified, = *initialization_expr* or => **NULL()** is used.

array_spec
: is a list of dimension bounds

entity_decl



a
: is an object name or function name. *array_spec* cannot be specified for a function name.

initial_value
: provides an initial value for the entity specified by the immediately preceding name

initialization_expr
: provides an initial value, by means of an initialization expression, for the entity specified by the immediately preceding name

=> **NULL()**
: provides the initial value for the pointer object

Rules

Within the context of a derived type definition:

- If => appears in a component initialization, the **POINTER** attribute must appear in the *attr_spec_list*.
- If = appears in a component initialization, the **POINTER** attribute cannot appear in the component *attr_spec_list*.
- The compiler will evaluate *initialization_expr* within the scoping unit of the type definition.

If => appears for a variable, the object must have the **POINTER** attribute.

If *initialization_expr* appears for a variable, the object cannot have the **POINTER** attribute.

Entities in type declaration statements are constrained by the rules of any attributes specified for the entities, as detailed in the corresponding attribute statements.

The type declaration statement overrides the implicit type rules in effect. You can use a type declaration statement that confirms the type of an intrinsic

function. The appearance of a generic or specific intrinsic function name in a type declaration statement does not cause the name to lose its intrinsic property.

An object cannot be initialized in a type declaration statement if it is a dummy argument, an allocatable array, a pointer, a function result, an object in blank common, an integer pointer, an external name, an intrinsic name, or an automatic object. Nor can an object be initialized if it has the **AUTOMATIC** attribute. The object may be initialized if it appears in a named common block in a block data program unit or if it appears in a named common block in a module.

In Fortran 95, a pointer can be initialized. Pointers can only be initialized by the use of **=> NULL()**.

The specification expression of an *array_spec* can be a nonconstant expression if the specification expression appears in an interface body or in the specification part of a subprogram. Any object being declared that uses this nonconstant expression and is not a dummy argument or a pointee is called an *automatic object*.

An attribute cannot be repeated in a given type declaration statement, nor can an entity be explicitly given the same attribute more than once in a scoping unit.

initialization_expr must be specified if the statement contains the **PARAMETER** attribute. If the entity you are declaring is a variable, and *initialization_expr* or **NULL()** is specified, the variable is initially defined. If the entity you are declaring is a derived type component, and *initialization_expr* or **NULL()** is specified, the derived type has default initialization. *a* becomes defined with the value determined by *initialization_expr*, in accordance with the rules for intrinsic assignment. If the entity is an array, its shape must be specified either in the type declaration statement or in a previous specification statement in the same scoping unit. A variable or variable subobject cannot be initialized more than once. If *a* is a variable, the presence of *initialization_expr* or => **NULL()** implies that *a* is a saved object, except for an object in a named common block. The initialization of an object could affect the fundamental storage class of an object.

An *array_spec* specified in the *entity_decl* takes precedence over the *array_spec* in the **DIMENSION** attribute.

An array function result that does not have the **POINTER** attribute must have an explicit-shape array specification.

If the entity declared is a function, it must not have an accessible explicit interface unless it is an intrinsic function.

If T or F, defined previously as the name of a constant, appears in a type declaration statement, it is no longer an abbreviated logical constant but the name of the named constant.

Examples

```
SUBROUTINE SUB
  DOUBLE COMPLEX, STATIC, DIMENSION(1) :: B
END SUBROUTINE
```

Related Information

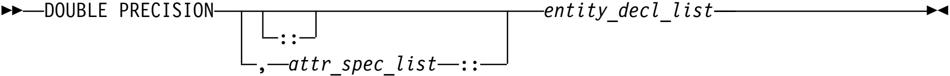
- "Complex" on page 33
- "Initialization Expressions" on page 87
- "How Type Is Determined" on page 52, for details on the implicit typing rules
- "Array Declarators" on page 65
- "Automatic Objects" on page 29
- "Storage Classes for Variables" on page 59
- "DATA" on page 277, for details on initial values

DOUBLE PRECISION

Purpose

A **DOUBLE PRECISION** type declaration statement specifies the attributes of objects and functions of type double precision. Initial values can be assigned to objects.

Format



where:

attr_spec
ALLOCATABLE **AUTOMATIC** **DIMENSION** (*array_spec*) **EXTERNAL** **INTENT** (*intent_spec*) **INTRINSIC** **OPTIONAL** **PARAMETER** **POINTER** **PRIVATE** **PUBLIC** **SAVE** **STATIC** **TARGET** **VOLATILE**

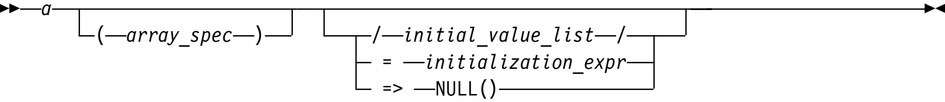
attr_spec
For detailed information on rules about a particular attribute, refer to the statement of the same name.

intent_spec
is either **IN**, **OUT**, or **INOUT**

:: is the double colon separator. It is required if attributes are specified, = *initialization_expr* or => **NULL()** is used.

array_spec
is a list of dimension bounds

entity_decl



a is an object name or function name. *array_spec* cannot be specified for a function name.

initial_value
provides an initial value for the entity specified by the immediately preceding name

initialization_expr
provides an initial value, by means of an initialization expression, for the entity specified by the immediately preceding name

=> **NULL()**
provides the initial value for the pointer object

Rules

Within the context of a derived type definition:

- If => appears in a component initialization, the **POINTER** attribute must appear in the *attr_spec_list*.
- If = appears in a component initialization, the **POINTER** attribute cannot appear in the component *attr_spec_list*.
- The compiler will evaluate *initialization_expr* within the scoping unit of the type definition.

If => appears for a variable, the object must have the **POINTER** attribute.

If *initialization_expr* appears for a variable, the object cannot have the **POINTER** attribute.

Entities in type declaration statements are constrained by the rules of any attributes specified for the entities, as detailed in the corresponding attribute statements.

The type declaration statement overrides the implicit type rules in effect. You can use a type declaration statement that confirms the type of an intrinsic function. The appearance of a generic or specific intrinsic function name in a type declaration statement does not cause the name to lose its intrinsic property.

An object cannot be initialized in a type declaration statement if it is a dummy argument, an allocatable array, a pointer, a function result, an object in blank common, an integer pointer, an external name, an intrinsic name, or an automatic object. Nor can an object be initialized if it has the **AUTOMATIC** attribute. The object may be initialized if it appears in a named common block in a block data program unit or if it appears in a named common block in a module.

In Fortran 95, a pointer can be initialized. Pointers can only be initialized by the use of **=> NULL()**.

The specification expression of an *array_spec* can be a nonconstant expression if the specification expression appears in an interface body or in the

specification part of a subprogram. Any object being declared that uses this nonconstant expression and is not a dummy argument or a pointee is called an *automatic object*.

An attribute cannot be repeated in a given type declaration statement, nor can an entity be explicitly given the same attribute more than once in a scoping unit.

initialization_expr must be specified if the statement contains the **PARAMETER** attribute. If the entity you are declaring is a variable, and *initialization_expr* or **NULL()** is specified, the variable is initially defined. If the entity you are declaring is a derived type component, and *initialization_expr* or **NULL()** is specified, the derived type has default initialization. *a* becomes defined with the value determined by *initialization_expr*, in accordance with the rules for intrinsic assignment. If the entity is an array, its shape must be specified either in the type declaration statement or in a previous specification statement in the same scoping unit. A variable or variable subobject cannot be initialized more than once. If *a* is a variable, the presence of *initialization_expr* or => **NULL()** implies that *a* is a saved object, except for an object in a named common block. The initialization of an object could affect the fundamental storage class of an object.

An *array_spec* specified in the *entity_decl* takes precedence over the *array_spec* in the **DIMENSION** attribute.

An array function result that does not have the **POINTER** attribute must have an explicit-shape array specification.

If the entity declared is a function, it must not have an accessible explicit interface unless it is an intrinsic function.

If `T` or `F`, defined previously as the name of a constant, appears in a type declaration statement, it is no longer an abbreviated logical constant but the name of the named constant.

Examples

```
DOUBLE PRECISION, POINTER :: PTR
DOUBLE PRECISION, TARGET  :: TAR
```

Related Information

- “Real” on page 30
- “Initialization Expressions” on page 87
- “How Type Is Determined” on page 52, for details on the implicit typing rules
- “Array Declarators” on page 65
- “Automatic Objects” on page 29
- “Storage Classes for Variables” on page 59

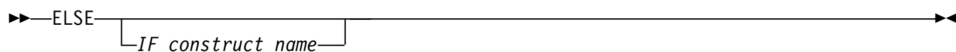
- "DATA" on page 277, for details on initial values

ELSE

Purpose

The **ELSE** statement is the first statement of the optional **ELSE** block within an **IF** construct.

Format



IF_construct_name
: is a name that identifies the **IF** construct

Format

Control branches to the **ELSE** block if every previous logical expression in the **IF** construct evaluates as false. The statement block of the **ELSE** block is executed and the **IF** construct is complete.

If you specify an *IF_construct_name*, it must be the same name that you specified in the block **IF** statement.

Examples

```
IF (A.GT.0) THEN
  B = B-A
ELSE            ! the next statement is executed if a<=0
  B = B+A
END IF
```

Related Information

- "IF Construct" on page 121
- "END (Construct)" on page 298, for details on the **END IF** statement
- "ELSE IF"

ELSE IF

Purpose

The **ELSE IF** statement is the first statement of an optional **ELSE IF** block within an **IF** construct.

Format

```
►►──ELSE IF──(──scalar_logical_expr──)──THEN──┬───────────────────┬──────────►◄
                                              └─IF_construct_name─┘
```

IF_construct_name
is a name that identifies the **IF** construct

Rules

scalar_logical_expr is evaluated if no previous logical expressions in the **IF** construct are evaluated as true. If *scalar_logical_expr* is true, the statement block that follows is executed and the **IF** construct is complete.

If you specify an *IF_construct_name*, it must be the same name that you specified in the block **IF** statement.

Examples

```
IF (I.EQ.1) THEN
    J=J-1
ELSE IF (I.EQ.2) THEN
    J=J-2
ELSE IF (I.EQ.3) THEN
    J=J-3
ELSE
    J=J-4
END IF
```

Related Information

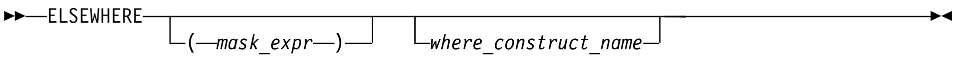
- “IF Construct” on page 121
- “END (Construct)” on page 298, for details on the **END IF** statement
- “ELSE” on page 294

ELSEWHERE

Purpose

The **ELSEWHERE** statement is the first statement of the optional **ELSEWHERE** or masked **ELSEWHERE** block within a **WHERE** construct.

Format



mask_expr is a logical array expression

where_construct_name
is a name that identifies a **WHERE** construct

Rules

A masked **ELSEWHERE** statement contains a *mask_expr*. See "Interpreting Masked Array Assignments" on page 108 for information on interpreting mask expressions. Each *mask_expr* in a **WHERE** construct must have the same shape.

If you specify a *where_construct_name*, it must be the same name that you specified on the **WHERE** construct statement.

ELSEWHERE and masked **ELSEWHERE** statements must not be branch target statements.

Examples

The following example shows a program that uses a simple masked **ELSEWHERE** statement to change the data in an array:

```
INTEGER ARR1(3, 3), ARR2(3, 3), FLAG(3, 3)

ARR1 = RESHAPE((/(I, I=1, 9)/), (/3, 3 /))
ARR2 = RESHAPE((/(I, I=9, 1, -1 /), (/3, 3 /))
FLAG = -99

! Data in arrays ARR1, ARR2, and FLAG at this point:
!
! ARR1 = |  1   4   7 |  ARR2 = |  9   6   3 |  FLAG = | -99 -99 -99 |
!        |  2   5   8 |         |  8   5   2 |         | -99 -99 -99 |
!        |  3   6   9 |         |  7   4   1 |         | -99 -99 -99 |

WHERE (ARR1 > ARR2)
  FLAG = 1
ELSEWHERE (ARR1 == ARR2)
  FLAG = 0
ELSEWHERE
  FLAG = -1
END WHERE

! Data in arrays ARR1, ARR2, and FLAG at this point:
!
! ARR1 = |  1   4   7 |  ARR2 = |  9   6   3 |  FLAG = | -1  -1   1 |
!        |  2   5   8 |         |  8   5   2 |         | -1   0   1 |
!        |  3   6   9 |         |  7   4   1 |         | -1   1   1 |
```

Related Information

- "WHERE Construct" on page 106
- "WHERE" on page 414
- "END (Construct)" on page 298, for details on the **END WHERE** statement

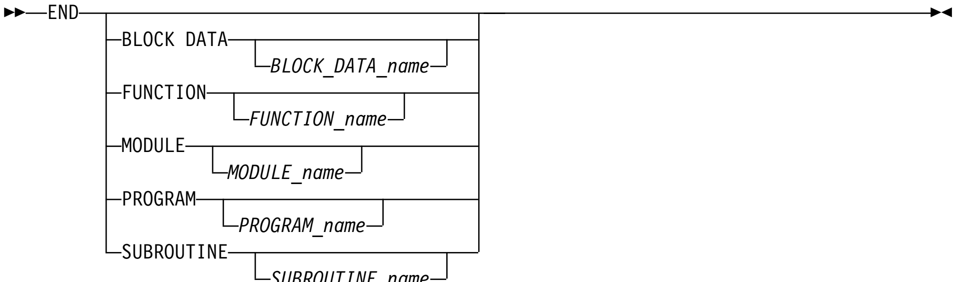
---

END

Purpose

An **END** statement indicates the end of a program unit or procedure.

Format



Rules

The **END** statement is the only required statement in a program unit.

For an internal subprogram or module subprogram, you must specify the **FUNCTION** or **SUBROUTINE** keyword on the **END** statement. For block data program units, external subprograms, the main program, modules and interface bodies, the corresponding keyword is optional.

The program name can be included in the **END PROGRAM** statement only if the optional **PROGRAM** statement is used and if the name is identical to the program name specified in the **PROGRAM** statement.

The block data name can be included in the **END BLOCK DATA** statement only if it is provided in the **BLOCK DATA** statement and if the name is identical to the block data name specified in the **BLOCK DATA** statement.

If a name is specified in an **END MODULE**, **END FUNCTION**, or **END SUBROUTINE** statement, it must be identical to the name specified in the corresponding **MODULE**, **FUNCTION**, or **SUBROUTINE** statement, respectively.

The **END**, **END FUNCTION**, **END PROGRAM**, and **END SUBROUTINE** statements are executable statements that can be branched to. In both fixed source form and Fortran 90 free source form formats, no other statement may follow the **END** statement on the same line. In fixed source form format, you cannot continue a program unit **END** statement, nor can a statement whose initial line appears to be a program unit **END** statement be continued.

The **END** statement of a main program terminates execution of the program. The **END** statement of a function or subroutine has the same effect as a

RETURN statement. An inline comment can appear on the same line as an **END** statement. Any comment line appearing after an **END** statement belongs to the next program unit.

Examples

```
PROGRAM TEST
  CALL SUB()
  CONTAINS
    SUBROUTINE SUB
   ⋮
    END SUBROUTINE    ! Reference to subroutine name SUB is optional
END PROGRAM TEST
```

Related Information

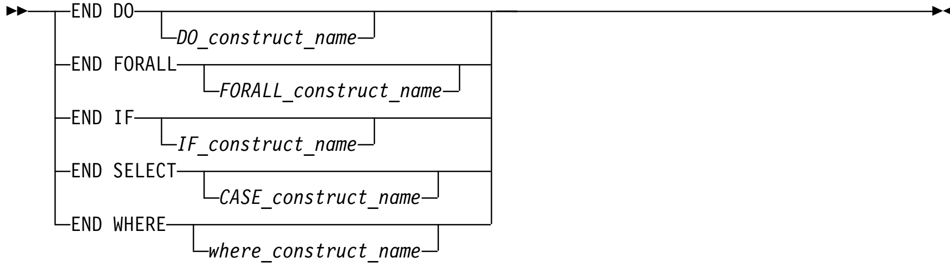
"Chapter 7. Program Units and Procedures" on page 133

END (Construct)

Purpose

The **END DO**, **END FORALL**, **END IF**, **END SELECT**, and **END WHERE** statements terminate **DO** (or **DO WHILE**), **FORALL**, **IF**, **CASE**, and **WHERE** constructs, respectively.

Format



DO_construct_name
: is a name that identifies a **DO** or **DO WHILE** construct

FORALL_construct_name
: is a name that identifies a **FORALL** construct

IF_construct_name
: is a name that identifies an **IF** construct

CASE_construct_name
: is a name that identifies a **CASE** construct

where_construct_name
is a name that identifies a **WHERE** construct

Rules

If you label the **END DO** statement, you can use it as the terminal statement of a labeled or unlabeled **DO** or **DO WHILE** construct. An **END DO** statement terminates the innermost **DO** or **DO WHILE** construct only. If a **DO** or **DO WHILE** statement does not specify a statement label, the terminal statement of the **DO** or **DO WHILE** construct must be an **END DO** statement.

You can branch to an **END DO**, **END IF**, or **END SELECT** statement from within the **DO** (or **DO WHILE**), **IF**, or **CASE** construct, respectively. An **END IF** statement can also be branched to from outside of the **IF** construct.

In Fortran 95, an **END IF** statement cannot be branched to from outside of the **IF** construct.

If you specify a construct name on the statement that begins the construct, the **END** statement that terminates the construct must have the same construct name. Conversely, if you do not specify a construct name on the statement that begins the construct, you must not specify a construct name on the **END** statement.

An **END WHERE** statement must not be a branch target statement.

Examples

```
INTEGER X(100,100)
DECR: DO WHILE (I.GT.0)
   ⋮
  IF (J.LT.K) THEN
   ⋮
  END IF                 ! Cannot reference a construct name
  I=I-1
END DO DECR              ! Reference to construct name DECR mandatory

END
```

The following example shows an invalid use of the *where_construct_name*:

```
BW: WHERE (A /= 0)
  B = B + 1
END WHERE EW      ! The where_construct_name on the END WHERE statement does not
                  ! match the where_construct_name on the WHERE statement
```

Related Information

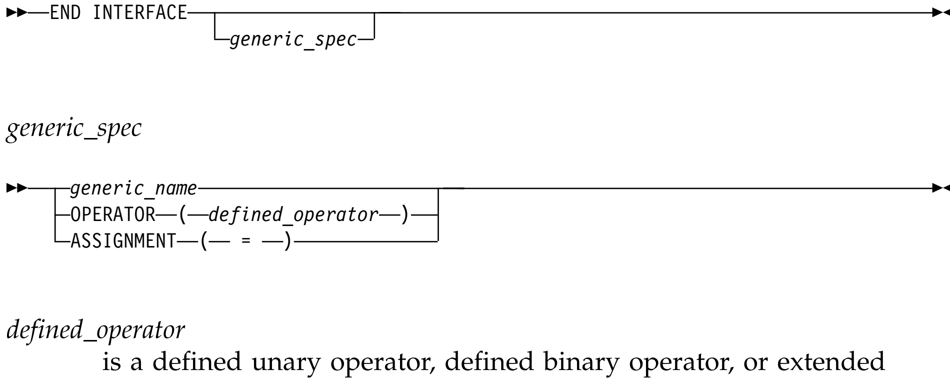
- "Chapter 6. Control" on page 121
- "DO" on page 284
- "FORALL" on page 311
- "FORALL (Construct)" on page 314
- "IF (Block)" on page 328
- "SELECT CASE" on page 390
- "WHERE" on page 414
- "Deleted Features" on page 735

END INTERFACE

Purpose

The **END INTERFACE** statement terminates a procedure interface block.

Format



defined_operator
: is a defined unary operator, defined binary operator, or extended intrinsic operator

Rules

Each **INTERFACE** statement must have a corresponding **END INTERFACE** statement.

If a *generic_spec* appears in an **END INTERFACE** statement, it must match the corresponding *generic_spec* in an **INTERFACE** statement.

An **END INTERFACE** statement without a *generic_spec* can match any **INTERFACE** statement, with or without a *generic_spec*.

If the *generic_spec* in an **END INTERFACE** statement is a *generic_name*, the *generic_spec* of the corresponding **INTERFACE** statement must be the same *generic_name*.

If the *generic_spec* in an **END INTERFACE** statement is an **OPERATOR**(*defined_operator*), the *generic_spec* of the corresponding **INTERFACE** statement must be the same **OPERATOR**(*defined_operator*).

If the *generic_spec* in an **END INTERFACE** statement is an **ASSIGNMENT**(=), the *generic_spec* for the corresponding **INTERFACE** statement must be the same **ASSIGNMENT**(=).

Examples

```
INTERFACE OPERATOR (.DETERMINANT.)
  FUNCTION DETERMINANT (X)
    INTENT(IN) X
    REAL X(50,50), DETERMINANT
  END FUNCTION
END INTERFACE
```

```
INTERFACE OPERATOR(.INVERSE.)
  FUNCTION INVERSE(Y)
    INTENT(IN) X
    REAL Y(50,50), INVERSE
  END FUNCTION
END INTERFACE OPERATOR(.INVERSE.)
```

Related Information

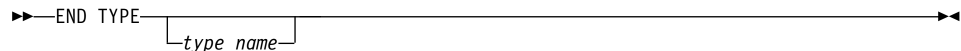
- “INTERFACE” on page 344
- “Interface Concepts” on page 142

END TYPE

Purpose

The **END TYPE** statement indicates the completion of a derived-type definition.

Format



Rules

If *type_name* is specified, it must match the *type_name* in the corresponding **Derived Type** statement.

If a label is specified on the **END TYPE** statement, the label belongs to the scoping unit of the derived-type definition.

Examples

```
TYPE A
  INTEGER :: B
  REAL :: C
END TYPE A
```

Related Information

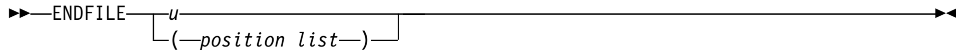
- "Derived Types" on page 39
- "Derived Type" on page 282

ENDFILE

Purpose

The **ENDFILE** statement writes an endfile record as the next record of an external file connected for sequential access. This record becomes the last record in the file.

Format



u
is an external unit identifier. The value of *u* must not be an asterisk or a Hollerith constant.

position_list
is a list that must contain one unit specifier ([**UNIT=**]*u*) and can also contain one of each of the other valid specifiers:

[UNIT=] *u*
is a unit specifier in which *u* must be an external unit identifier whose value is not an asterisk. An external unit identifier refers to an external file that is represented by a scalar integer expression, whose value is in the range 1 through 2147483647. If the optional characters **UNIT=** are omitted, *u* must be the first item in *position_list*.

IOSTAT= *ios*
is an input/output status specifier that specifies the status of the input/output operation. *ios* is a scalar variable of type **INTEGER(4)** or default integer. When the **ENDFILE** statement finishes executing, *ios* is defined with:
- A zero value if no error condition occurs
- A positive value if an error occurs.

ERR= *stmt_label*
is an error specifier that specifies the statement label of an executable

statement in the same scoping unit to which control is to transfer in the case of an error. Coding the **ERR=** specifier suppresses error messages.

Rules

If the unit is not connected, an implicit **OPEN** specifying sequential access is performed to a default file named **fort.***n*, where *n* is the value of *u* with leading zeros removed.

If two **ENDFILE** statements are executed for the same file without an intervening **REWIND** or **BACKSPACE** statement, the second **ENDFILE** statement is ignored.

If the **ERR=** and **IOSTAT=** specifiers are set and an error is encountered, transfer is made to the statement specified by the **ERR=** specifier and a positive integer value is assigned to *ios*.

If **IOSTAT=** and **ERR=** are not specified,
- The program stops if a severe error is encountered.
- The program continues to the next statement if a recoverable error is encountered and the **ERR_RECOVERY** run-time option is set to **YES**. If the option is set to **NO**, the program stops.

Examples

```
ENDFILE 12
ENDFILE (IOSTAT=IOSS,UNIT=11)
```

Related Information

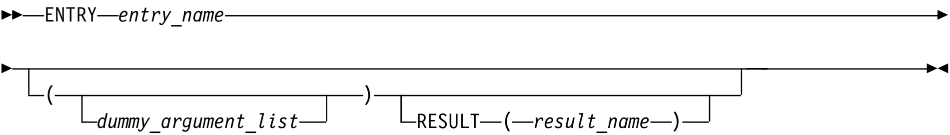
- “Conditions and IOSTAT Values” on page 193
- “Chapter 8. Input/Output Concepts” on page 183
- ″Setting Runtime Options for Input/Output″ in the *User's Guide*

ENTRY

Purpose

A function subprogram or subroutine subprogram has a primary entry point that is established through the **SUBROUTINE** or **FUNCTION** statement. The **ENTRY** statement establishes an alternate entry point for an external subprogram or a module subprogram.

Format



entry_name
: is the name of an entry point in a function subprogram or subroutine subprogram

Rules

The **ENTRY** statement cannot appear in a main program, block data program unit, internal subprogram, **IF** construct, **DO** construct, **CASE** construct, derived-type definition, or interface block.

The **ENTRY** statement cannot appear in a **CRITICAL**, **MASTER**, **PARALLEL**, or **PARALLEL SECTIONS** construct.

An **ENTRY** statement can appear anywhere after the **FUNCTION** or **SUBROUTINE** statement (and after any **USE** statements) of an external or module subprogram, except in a statement block within a control construct, in a derived-type definition, or in an interface block. **ENTRY** statements are nonexecutable and do not affect control sequencing during the execution of a subprogram.

The result variable is *result_name*, if specified; otherwise, it is *entry_name*. If the characteristics of the **ENTRY** statement's result variable are the same as those of the **FUNCTION** statement's result variable, the result variables identify the same variable, even though they can have different names. Otherwise, they are storage-associated and must be all nonpointer scalars of intrinsic (noncharacter) type. *result_name* can be the same as the result variable name specified for the **FUNCTION** statement or another **ENTRY** statement.

The result variable cannot be specified in a **COMMON**, **DATA**, integer **POINTER**, or **EQUIVALENCE** statement, nor can it have the **ALLOCATABLE**, **PARAMETER**, **INTENT**, **OPTIONAL**, **SAVE**, or **VOLATILE** attributes. The **STATIC** and **AUTOMATIC** attributes can be specified only if the result variable is not an array or a pointer and is not of character or derived type.

If the **RESULT** keyword is specified, the **ENTRY** statement must be within a function subprogram, *entry_name* must not appear in any specification statement in the scope of the function subprogram, and *result_name* cannot be the same as *entry_name*.

A result variable cannot be initialized in a type declaration statement.

The entry name in an external subprogram is a global entity; an entry name in a module subprogram is not a global entity. An interface for an entry can appear in an interface block only when the entry name is used as the procedure name in an interface body.

In a function subprogram, *entry_name* identifies an external or module function that can be referenced as a function from the calling procedure. In a subroutine subprogram, *entry_name* identifies a subroutine and can be referenced as a subroutine from the calling procedure. When the reference is made, execution begins with the first executable statement following the **ENTRY** statement.

The result variable must be defined prior to exiting from the function, when the function is invoked through that entry.

A name in the *dummy_argument_list* must not appear in the following places:
- In an executable statement preceding the **ENTRY** statement unless it also appears in a **FUNCTION**, **SUBROUTINE**, or **ENTRY** statement that precedes the executable statement.
- In the expression of a statement function statement, unless the name is also a dummy argument of the statement function, appears in a **FUNCTION** or **SUBROUTINE** statement, or appears in an **ENTRY** statement that precedes the statement function statement.

The order, number, type, and kind type parameters of the dummy arguments can differ from those of the **FUNCTION** or **SUBROUTINE** statement, or other **ENTRY** statements.

If a dummy argument is used in a specification expression to specify an array bound or character length of an object, you can only specify the object in a statement that is executed during a procedure reference if the dummy argument is present and appears in the dummy argument list of the procedure name referenced.

Recursion

An **ENTRY** statement can reference itself directly only if the subprogram statement specifies **RECURSIVE** and the **ENTRY** statement specifies **RESULT**. The entry procedure then has an explicit interface within the subprogram. The **RESULT** clause is not required for an entry to reference itself indirectly.

Elemental subprograms can have **ENTRY** statements, but the **ENTRY** statement cannot have the **ELEMENTAL** prefix. The procedure defined by the **ENTRY** statement is elemental if the **ELEMENTAL** prefix is specified in the **SUBROUTINE** or **FUNCTION** statement.

If *entry_name* is of type character, its length cannot be an asterisk if the function is recursive.

You can also call external procedures recursively when you specify the **-qrecur** compiler option, although XL Fortran disregards this option if a procedure specifies either the **RECURSIVE** or **RESULT** keyword.

Examples

```
RECURSIVE FUNCTION FNC() RESULT (RES)
   ⋮
  ENTRY ENT () RESULT (RES)          ! The result variable name can be
                                     ! the same as for the function
   ⋮
END FUNCTION
```

Related Information

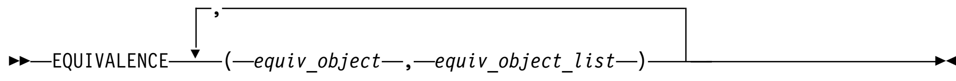
- "FUNCTION" on page 320
- "SUBROUTINE" on page 396
- "Recursion" on page 175
- "Dummy Arguments" on page 162
- "-qrecur Option" in the *User's Guide*

EQUIVALENCE

Purpose

The **EQUIVALENCE** statement specifies that two or more objects in a scoping unit are to share the same storage.

Format



equiv_object
: is a variable name, array element, or substring. Any subscript or substring expression must be an integer initialization expression.

Rules

equiv_object must not be a target, pointer, dummy argument, function name, pointee, entry name, result name, structure component, named constant,

automatic data object, allocatable array, object of nonsequence derived type, object of sequence derived type that contains a pointer in the structure, or a subobject of any of these.

Because all items named within a pair of parentheses have the same first storage unit, they become associated. This is called *equivalence association*. It may cause the association of other items as well.

You can specify default initialization for a storage unit that is storage associated. However, the objects or subobjects supplying the default initialization must be of the same type. They must also be of the same type parameters and supply the same value for the storage unit.

If you specify an array element in an **EQUIVALENCE** statement, the number of subscript quantities cannot exceed the number of dimensions in the array. If you specify a multidimensional array using an array element with a single subscript *n*, the *n* element in the array's storage sequence is specified. In all other cases, XL Fortran replaces any missing subscript with the lower bound of the corresponding dimension of the array. A nonzero-sized array without a subscript refers to the first element of the array.

If *equiv_object* is of derived type, it must be of a sequence derived type.

You can equivalence an object of sequence derived type with any other object of sequence derived type or intrinsic data type provided that the object is allowed in an **EQUIVALENCE** statement.

In XL Fortran, associated items can be of any intrinsic type or of sequence derived type. If they are, the **EQUIVALENCE** statement does not cause type conversion.

The lengths of associated items do not have to be equal.

Any zero-sized items are storage-associated with one another and with the first storage unit of any nonzero-sized sequences.

An **EQUIVALENCE** statement cannot associate the storage sequences of two different common blocks. It must not specify that the same storage unit is to occur more than once in a storage sequence. An **EQUIVALENCE** statement must not contradict itself or any previously established associations caused by an **EQUIVALENCE** statement.

You can cause names not in common blocks to share storage with a name in a common block using the **EQUIVALENCE** statement.

You can extend a common block by using an **EQUIVALENCE** statement, but only by adding beyond the last entry, not before the first entry. For example, if the variable that you associate to a variable in a common block, using the **EQUIVALENCE** statement, is an element of an array, the implicit association of the rest of the elements of the array can extend the size of the common block.

Examples

```
DOUBLE PRECISION A(3)
REAL B(5)
EQUIVALENCE (A,B(3))
```

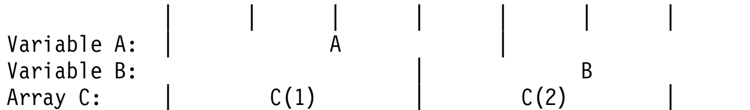
Association of storage units:

```
           |      |      |      |      |      |      |      |
Array A:                 |    A(1)     |    A(2)     |    A(3)     |
Array B:   | B(1) | B(2) | B(3) | B(4) | B(5) |
```

This example shows how association of two items can result in further association.

```
AUTOMATIC A
CHARACTER A*4,B*4,C(2)*3
EQUIVALENCE (A,C(1)),(B,C(2))
```

Association of storage units:



Because XL Fortran associates both `A` and `B` with `C`, `A` and `B` become associated with each other, and they all have the automatic storage class.

```
INTEGER(4)   G(2,-1:2,-3:2)
REAL(4)      H(3,1:3,2:3)
EQUIVALENCE  (G(2),H(1,1))   ! G(2) is G(2,-1,-3)
                             ! H(1,1) is H(1,1,2)
```

Related Information

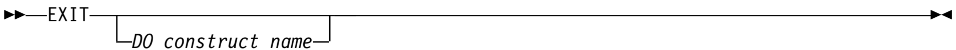
- "Storage Classes for Variables" on page 59
- "Definition Status of Variables" on page 53

EXIT

Purpose

The **EXIT** statement terminates execution of a **DO** construct or **DO WHILE** construct before the construct completes all of its iterations.

Format



DO_construct_name
is the name of the **DO** or **DO WHILE** construct

Rules

The **EXIT** statement is placed within a **DO** or **DO WHILE** construct and belongs to the **DO** or **DO WHILE** construct specified by *DO_construct_name* or, if not specified, by the **DO** or **DO WHILE** construct that immediately surrounds it. When a *DO_construct_name* is specified, the **EXIT** statement must be in the range of that construct.

When the **EXIT** statement is executed, the **DO** or **DO WHILE** construct that the **EXIT** statement belongs to becomes inactive. If the **EXIT** statement is nested in any other **DO** or **DO WHILE** constructs, they also become inactive. Any **DO** variable present retains its last defined value. If the **DO** construct has no construct control, it will iterate infinitely unless it becomes inactive. The **EXIT** statement can be used to make the construct inactive.

An **EXIT** statement can have a statement label; it cannot be used as the labeled statement that terminates a **DO** or **DO WHILE** construct.

Examples

```
      LOOP1: DO I = 1, 20
         N = N + 1
10       IF (N > NMAX) EXIT LOOP1          ! EXIT from LOOP1

         LOOP2: DO WHILE (K==1)
            KMAX = KMAX - 1
20          IF (K > KMAX) EXIT             ! EXIT from LOOP2
         END DO LOOP2

         LOOP3:  DO J = 1, 10
            N = N + 1
30          IF (N > NMAX) EXIT LOOP1       ! EXIT from LOOP1
            EXIT LOOP3                     ! EXIT from LOOP3
         END DO LOOP3

      END DO LOOP1
```

Related Information

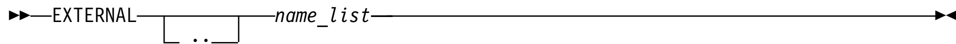
- "DO Construct" on page 126
- "DO WHILE Construct" on page 130

EXTERNAL

Purpose

The **EXTERNAL** attribute specifies that a name represents an external procedure, a dummy procedure, or a block data program unit. A procedure name with the **EXTERNAL** attribute can be used as an actual argument.

Format



name is the name of an external procedure, dummy procedure, or **BLOCK DATA** program unit

Rules

If an external procedure name or dummy argument name is used as an actual argument, it must be declared with the **EXTERNAL** attribute or by an interface block in the scoping unit, but may not appear in both.

If an intrinsic procedure name is specified with the **EXTERNAL** attribute in a scoping unit, the name becomes the name of a user-defined external procedure. Therefore, you cannot invoke that intrinsic procedure by that name from that scoping unit.

You can specify a name to have the **EXTERNAL** attribute appear only once in a scoping unit.

A name in an **EXTERNAL** statement must not also be specified as a specific procedure name in an interface block in the scoping unit.

Attributes Compatible with the EXTERNAL Attribute

- OPTIONAL
- PRIVATE
- PUBLIC

Examples

```
PROGRAM MAIN
  EXTERNAL AAA
  CALL SUB(AAA)         ! Procedure AAA is passed to SUB
END

SUBROUTINE SUB(ARG)
  CALL ARG()            ! This results in a call to AAA
END SUBROUTINE
```

Related Information

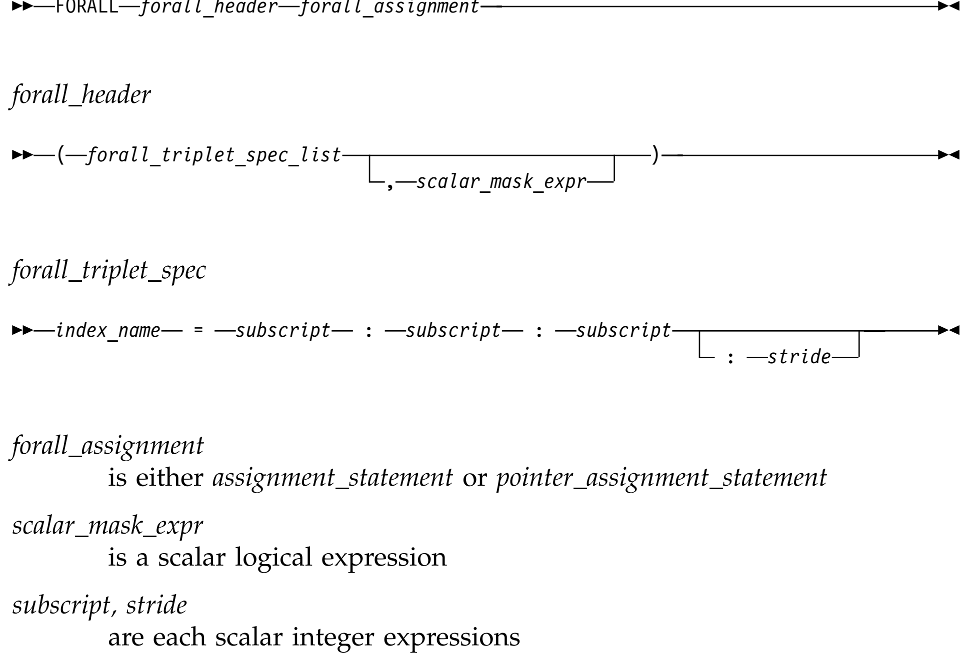
- “Procedures as Dummy Arguments” on page 171
- Item 4 under “Appendix A. Compatibility Across Standards” on page 731

FORALL

Purpose

The **FORALL** statement performs assignment to groups of subobjects, especially array elements. Unlike the **WHERE** statement, assignment can be performed on an elemental level rather than on an array level. The **FORALL** statement also allows pointer assignment.

Format



forall_assignment
: is either *assignment_statement* or *pointer_assignment_statement*

scalar_mask_expr
: is a scalar logical expression

subscript, stride
: are each scalar integer expressions

Rules

Only pure procedures can be referenced in the mask expression of *forall_header* and in a *forall_assignment* (including one referenced by a defined operation or assignment).

index_name must be a scalar integer variable. It is also a statement entity; that is, it does not affect and is not affected by other entities in the scoping unit.

In *forall_triplet_spec_list*, neither a *subscript* nor a *stride* can contain a reference to any *index_name* in the *forall_triplet_spec_list*. Evaluation of any expression in *forall_header* must not affect evaluation of any other expression in *forall_header*.

Given the *forall_triplet_spec*

```
index1 = s1:s2:s3
```

the maximum number of index values is determined by:

```
max = INT((s2-s1+s3)/s3)
```

If the stride (*s3* above) is not specified, a value of 1 is assumed. If *max* ≤ 0 for any index, *forall_assignment* is not executed. For example,

```
index1 = 2:10:3    !  The index values are 2,5,8.
                      max = INT((10-2+3)/3) = 3.

index2 = 6:2:-1    !  The index values are 6,5,4,3,2.
index2 = 6:2       !  No index values.
```

If the mask expression is omitted, a value of .TRUE. is assumed.

No atomic object can be assigned to more than once. Assignment to a nonatomic object assigns to all subobjects or associates targets with all subobjects.

Interpreting the FORALL Statement

1. Evaluate the *subscript* and *stride* expressions for each *forall_triplet_spec* in any order. All possible pairings of *index_name* values form the set of combinations. For example, given the following statement:

   ```
   FORALL (I=1:3,J=4:5) A(I,J) = A(J,I)
   ```

   The set of combinations of I and J is:

   ```
   {(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)}
   ```

   The **-1** and **-qnozerosize** compiler options do not affect this step.
2. Evaluate the *scalar_mask_expr* for the set of combinations, in any order, producing a set of active combinations (those for which *scalar_mask_expr* evaluated to .TRUE.). For example, if the mask (I+J.NE.6) is applied to the above set, the set of active combinations is:

   ```
   {(1,4),(2,5),(3,4),(3,5)}
   ```
3. For *assignment_statement*, evaluate, in any order, all values in the right-hand side *expression* and all subscripts, strides, and substring bounds in the left-hand side *variable* for all active combinations of *index_name* values.

   For *pointer_assignment*, determine, in any order, what will be the targets of the pointer assignment and evaluate all subscripts, strides, and substring

bounds in the pointer for all active combinations of *index_name* values. Whether or not the target is a pointer, the determination of the target does not include evaluation of its value.

4. For *assignment_statement*, assign, in any order, the computed *expression* values to the corresponding *variable* entities for all active combinations of *index_name* values.

   For *pointer_assignment*, associate, in any order, all targets with the corresponding pointer entities for all active combinations of *index_name* values.

Loop Parallelization

The **FORALL** statement and **FORALL** construct are designed to allow for parallelization of assignment statements. When executing an assignment statement in a **FORALL**, the assignment of an object will not interfere with the assignment of another object. In the next example, the assignments to elements of A can be executed in any order without changing the results:

```
FORALL (I=1:3,J=1:3) A(I,J)=A(J,I)
```

The **INDEPENDENT** directive asserts that each iteration of a **DO** loop or each operation in a **FORALL** statement or **FORALL** construct can be executed in any order without affecting the semantics of the program. The operations in a **FORALL** statement or **FORALL** construct are defined as:

- The evaluation of *mask*
- The evaluation of the right-hand side and/or left-hand side indexes
- The evaluation of assignments

Thus, the following loop,

```
        INTEGER, DIMENSION(2000) :: A,B,C
!IBM*  INDEPENDENT
        DO I = 1, 1999, 2
          A(I) = A(I+1)
        END DO
```

is semantically equivalent to the following array assignment:

```
        INTEGER, DIMENSION(2000) :: A,B,C
        A(1:1999:2) = A(2:2000:2)
```

Tip

If it is possible and beneficial to make a specific **FORALL** parallel, specify the **INDEPENDENT** directive before the **FORALL** statement. Because XL Fortran may not always be able to determine whether it is legal to parallelize a **FORALL**, the **INDEPENDENT** directive provides an assertion that it is legal.

Examples

```
INTEGER A(1000,1000), B(200)
I=17
FORALL (I=1:1000,J=1:1000,I.NE.J) A(I,J)=A(J,I)
PRINT *, I    ! The value 17 is printed because the I
              ! in the FORALL has statement scope.
FORALL (N=1:200:2) B(N)=B(N+1)
END
```

Related Information

- "Intrinsic Assignment" on page 103
- "Pointer Assignment" on page 117
- "FORALL Construct" on page 113
- "INDEPENDENT" on page 466
- "Statement and Construct Entities" on page 136

FORALL (Construct)

Purpose

The **FORALL (Construct)** statement is the first statement of the **FORALL** construct.

Format

```
>>-+------------------------+--FORALL--+---------------+--------><
   '-FORALL_construct_name-:-'         '-forall_header-'
```

forall_header

```
>>-(--forall_triplet_spec_list--+--------------------+--)-------><
                                '-,--scalar_mask_expr-'
```

forall_triplet_spec

```
>>--index_name-- = --subscript-- : --subscript--+-------------+--><
                                                '- : --stride-'
```

scalar_mask_expr
: is a scalar logical expression

subscript, stride
: are both scalar integer expressions

Rules

Any procedures that are referenced in the mask expression of *forall_header* (including one referenced by a defined operation or assignment) must be pure.

The *index_name* must be a scalar integer variable. The scope of *index_name* is the whole **FORALL** construct.

In *forall_triplet_spec_list*, neither a *subscript* nor a *stride* can contain a reference to any *index_name* in the *forall_triplet_spec_list*. Evaluation of any expression in *forall_header* must not affect evaluation of any other expression in *forall_header*.

Given the following *forall_triplet_spec*:

```
index1 = s1:s2:s3
```

The maximum number of index values is determined by:

```
max = INT((s2-s1+s3)/s3)
```

If the stride (*s3* above) is not specified, a value of 1 is assumed. If *max* ≤ 0 for any index, *forall_assignment* is not executed. For example:

```
index1 = 2:10:3    !  The index values are 2,5,8.
                   !  max = floor(((10-2)/3)+1) = 3.

index2 = 6:2:-1    !  The index values are 6,5,4,3,2.
index2 = 6:2       !  No index values.
```

If the mask expression is omitted, a value of `.TRUE.` is assumed.

Examples

```
POSITIVE: FORALL (X=1:100,A(X)>0)
  I(X)=I(X)+J(X)
  J(X)=J(X)-I(X+1)
END FORALL POSITIVE
```

Related Information

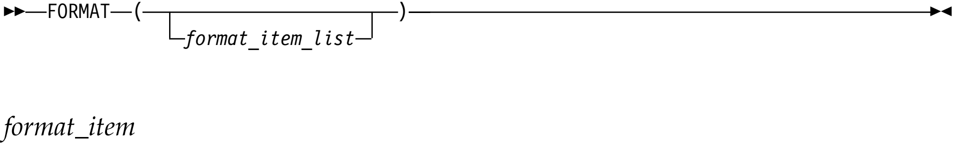
- "END (Construct)" on page 298
- "FORALL Construct" on page 113
- "Statement and Construct Entities" on page 136

FORMAT

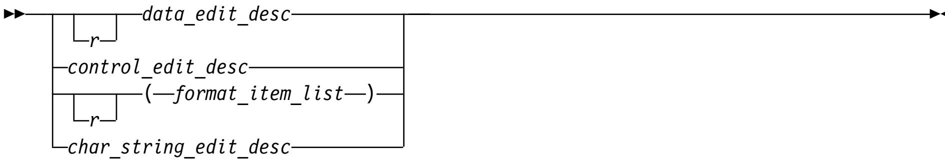
Purpose

The **FORMAT** statement provides format specifications for input/output statements.

Format



format_item



r
is an unsigned, positive, integer literal constant that cannot specify a kind type parameter, or it is a scalar integer expression enclosed by angle brackets (< and >). It is called a repeat specification. It specifies the number of times to repeat the *format_item_list* or the *data_edit_desc*. The default is 1.

data_edit_desc
is a data edit descriptor

control_edit_desc
is a control edit descriptor

char_string_edit_desc
is a character string edit descriptor

Data Edit Descriptors

Forms	Use	Page
A **A***w*	Edits character values	205
B*w* **B***w.m*	Edits binary values	206
E*w.d* **E***w.d***E***e* **E***w.d***D***e* **E***w.d***Q***e* **D***w.d* **EN***w.d* **EN***w.d***E***e* **ES***w.d* **ES***w.d***E***e* **Q***w.d*	Edits real and complex numbers with exponents	207

Forms	Use	Page
F*w.d*	Edits real and complex numbers without exponents	211
G*w.d* **G***w.d***E***e* **G***w.d***D***e* **G***w.d***Q***e*	Edits data fields of any intrinsic type, with the output format adapting to the type of the data and, if the data is of type real, the magnitude of the data	212
I*w* **I***w.m*	Edits integer numbers	214
L*w*	Edits logical values	215
O*w* **O***w.m*	Edits octal values	216
Q	Returns the count of characters remaining in an input record	217
Z*w* **Z***w.m*	Edits hexadecimal values	219

where:

w specifies the width of a field, including all blanks. It must be positive, except that it can be zero for **I**, **B**, **O**, **Z**, and **F** edit descriptors on output.

m specifies the number of digits to be printed

d specifies the number of digits to the right of the decimal point

e specifies the number of digits in the exponent field

w, *m*, *d*, and *e* can be:
- An unsigned integer literal constant
- A scalar integer expression enclosed by angle brackets (< and >). See “Variable Format Expressions” on page 320 for details.

You cannot specify kind parameters for *w*, *m*, *d*, or *e*.

Note:

There are two types of **Q** data edit descriptor (**Q***w.d* and **Q**):

extended precision Q
is the **Q** edit descriptor whose syntax is **Q***w.d*

character count Q
is the **Q** edit descriptor whose syntax is **Q**

Control Edit Descriptors

Forms	Use	Page
/ *r* /	Specifies the end of data transfer on the current record	220
:	Specifies the end of format control if there are no more items in the input/output list	221
$	Suppresses end-of-record in output	221
BN	Ignores nonleading blanks in numeric input fields	222
BZ	Interprets nonleading blanks in numeric input fields as zeros	222
*k***P**	Specifies a scale factor for real and complex items	224
S SS	Specifies that plus signs are not to be written	224
SP	Specifies that plus signs are to be written	224
T*c*	Specifies the absolute position in a record from which, or to which, the next character is transferred	225
TL*c*	Specifies the relative position (backward from the current position in a record) from which, or to which, the next character is transferred	225
TR*c*	Specifies the relative position (forward from the current position in a record) from which, or to which, the next character is transferred	225
*o***X**	Specifies the relative position (forward from the current position in a record) from which, or to which, the next character is transferred	225

where:

r is a repeat specifier. It is an unsigned, positive, integer literal constant.

k specifies the scale factor to be used. It is an optionally signed, integer literal constant.

c specifies the character position in a record. It is an unsigned, nonzero, integer literal constant.

o is the relative character position in a record. It is an unsigned, nonzero, integer literal constant.

r, *k*, *c*, and *o* can also be expressed as an arithmetic expression enclosed by angle brackets (< and >) that evaluates into an integer value.

Kind type parameters cannot be specified for *r*, *k*, *c*, or *o*.

Character String Edit Descriptors

Forms	Use	Page
*n***H***str*	Outputs a character string (str)	223
'*str*' "*str*"	Outputs a character string (*str*)	221

n
: is the number of characters in a literal field. It is an unsigned, positive, integer literal constant. Blanks are included in character count. A kind type parameter cannot be specified.

Rules

When a format identifier in a formatted **READ**, **WRITE**, or **PRINT** statement is a statement label or a variable that is assigned a statement label, the statement label identifies a **FORMAT** statement.

The **FORMAT** statement must have a statement label. **FORMAT** statements cannot appear in block data program units, interface blocks, the scope of a module, or derived-type definitions.

Commas separate edit descriptors. You can omit the comma between a **P** edit descriptor and an **F**, **E**, **EN**, **ES**, **D**, **G**, or **Q** (both extended precision and character count) edit descriptor immediately following it, before a slash edit descriptor when the optional repeat specification is not present, after a slash edit descriptor, and before or after a colon edit descriptor.

FORMAT specifications can also be given as character expressions in input/output statements.

XL Fortran treats uppercase and lowercase characters in format specifications the same, except in character string edit descriptors.

Character Format Specification

When a format identifier (page 376) in a formatted **READ**, **WRITE**, or **PRINT** statement is a character array name or character expression, the value of the array or expression is a character format specification.

If the format identifier is a character array element name, the format specification must be completely contained within the array element. If the format identifier is a character array name, the format specification can continue beyond the first element into following consecutive elements.

Blanks can precede the format specification. Character data can follow the right parenthesis that ends the format specification without affecting the format specification.

Variable Format Expressions: Wherever an integer constant is required by an edit descriptor, you can specify an integer expression in a **FORMAT** statement. The integer expression must be enclosed by angle brackets (**<** and **>**). You cannot use a sign outside of a variable format expression. The following are valid format specifications:

```
      WRITE(6,20) INT1
20    FORMAT(I<MAX(20,5)>)

      WRITE(6,FMT=30) INT2, INT3
30    FORMAT(I<J+K>,I<2*M>)
```

The integer expression can be any valid Fortran expression, including function calls and references to dummy arguments, with the following restrictions:
- Expressions cannot be used with the **H** edit descriptor
- Expressions cannot contain graphical relational operators.

The value of the expression is reevaluated each time an input/output item is processed during the execution of the **READ**, **WRITE**, or **PRINT** statement.

Examples

```
      CHARACTER*32 CHARVAR
      CHARVAR="('integer: ',I2,'  binary: ',B8)"  ! Character format specification
      M = 56
      J = 1                                       !     OUTPUT:
      X = 2355.95843                              !
      WRITE (6,770) M,X                           !  56   2355.96
      WRITE (6,CHARVAR) M,M                       ! integer: 56  binary: 00111000
      WRITE (6,880) J,M                           !  1
                                                  ! 56
770   FORMAT(I3, 2F10.2)
880   FORMAT(I<J+1>)
      END
```

Related Information

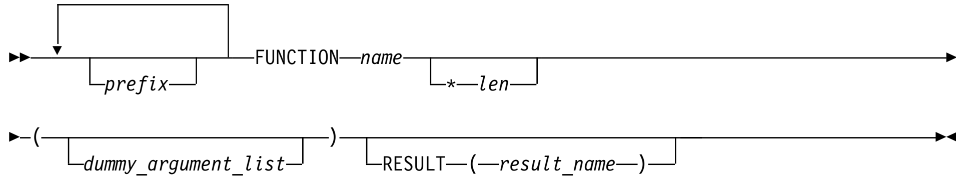
- “Chapter 9. Input/Output Formatting” on page 201
- “PRINT” on page 368
- “READ” on page 374
- “WRITE” on page 416

FUNCTION

Purpose

The **FUNCTION** statement is the first statement of a function subprogram.

Format



prefix is one of the following:
type_spec
RECURSIVE
PURE
ELEMENTAL

type_spec
specifies the type and type parameters of the function result. See "Type Declaration" on page 402 for details about *type_spec*.

name is the name of the function subprogram

len is an unsigned, positive, integer literal constant that cannot specify a kind type parameter. It represents the permissible length specifications for its associated type. It can be included only when the type is specified. The type cannot be **DOUBLE PRECISION**, **DOUBLE COMPLEX**, **BYTE**, or a derived type.

Rules

At most one of each kind of *prefix* can be specified.

The type and type parameters of the function result can be specified by either *type_spec* or by declaring the result variable in the declaration part of the function subprogram, but not by both. If they are not specified at all, the implicit typing rules are in effect. A length specifier cannot be specified by both *type_spec* and *len*.

If **RESULT** is specified, *result_name* becomes the function result variable. *name* must not be declared in any specification statement in the subprogram, although it can be referenced. *result_name* must not be the same as *name*. If **RESULT** is not specified, *name* becomes the function result variable.

If the result variable is an array or pointer, the **DIMENSION** or **POINTER** attributes, respectively, must be specified within the function body.

If the function result is a pointer, the shape of the result variable determines the shape of the value returned by the function. If the result variable is a

pointer, the function must either associate a target with the pointer or define the association status of the pointer as disassociated.

If the result variable is not a pointer, the function must define its value.

If the name of an external function is of derived type, the derived type must be a sequence derived type if the type is not use-associated or host-associated.

The function result variable must not appear within a variable format expression, nor can it be specified in a **COMMON**, **DATA**, integer **POINTER**, or **EQUIVALENCE** statement, nor can it have the **ALLOCATABLE**, **PARAMETER**, **INTENT**, **OPTIONAL**, or **SAVE** attributes. The **AUTOMATIC** or **STATIC** attributes can be specified if it is not an array or pointer, or if it is not of character or derived type.

The function result variable is associated with any entry procedure result variables. This is called entry association. The definition of any of these result variables becomes the definition of all the associated variables having that same type, and is the value of the function regardless of the entry point.

If the function subprogram contains entry procedures, the result variables are not required to be of the same type unless the type is of character or derived type, or if the variables have the **POINTER** attribute, or if they are not scalars. The variable whose name is used to reference the function must be in a defined state when a **RETURN** or **END** statement is executed in the subprogram. An associated variable of a different type must not become defined during the execution of the function reference, unless an associated variable of the same type redefines it later during execution of the subprogram.

Recursion

The **RECURSIVE** keyword must be specified if, directly or indirectly:
- The function invokes itself
- The function invokes a function defined by an **ENTRY** statement in the same subprogram
- An entry procedure in the same subprogram invokes itself
- An entry procedure in the same subprogram invokes another entry procedure in the same subprogram
- An entry procedure in the same subprogram invokes the subprogram defined by the **FUNCTION** statement.

A function that directly invokes itself requires that both the **RECURSIVE** and **RESULT** keywords be specified. The presence of both keywords makes the procedure interface explicit within the subprogram.

If *name* is of type character, its length cannot be an asterisk if the function is recursive.

If **RECURSIVE** is specified, the result variable has a default storage class of automatic.

You can also call external procedures recursively when you specify the **-qrecur** compiler option, although XL Fortran disregards this option if the **FUNCTION** statement specifies either **RECURSIVE** or **RESULT**.

Elemental Procedures

For elemental procedures, the keyword **ELEMENTAL** must be specified. If the **ELEMENTAL** keyword is specified, the **RECURSIVE** keyword cannot be specified.

Examples

```
RECURSIVE FUNCTION FACTORIAL (N) RESULT (RES)
  INTEGER RES
  IF (N.EQ.0) THEN
    RES=1
  ELSE
    RES=N*FACTORIAL(N-1)
  END IF
END FUNCTION FACTORIAL
```

```
PROGRAM P
  INTERFACE OPERATOR (.PERMUTATION.)
    ELEMENTAL FUNCTION MYPERMUTATION(ARR1,ARR2)
      INTEGER :: MYPERMUTATION
      INTEGER, INTENT(IN) :: ARR1,ARR2
    END FUNCTION MYPERMUTATION
  END INTERFACE

  INTEGER PERMVEC(100,150),N(100,150),K(100,150)
  ...
  PERMVEC = N .PERMUTATION. K
  ...
END
```

Related Information

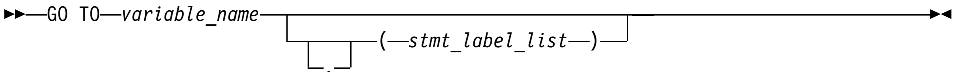
- “Function and Subroutine Subprograms” on page 157
- “ENTRY” on page 303
- “Function Reference” on page 159
- “Dummy Arguments” on page 162
- “Statement Function” on page 392
- “Recursion” on page 175
- "-qrecur Option" in the *User's Guide*
- “Pure Procedures” on page 176
- “Elemental Procedures” on page 178

GO TO (Assigned)

Purpose

The assigned **GO TO** statement transfers program control to an executable statement, whose statement label is designated in an **ASSIGN** statement.

Format



variable_name
: is a scalar variable name of type **INTEGER(4)** or **INTEGER(8)** that you have assigned a statement label to in an **ASSIGN** statement.

stmt_label
: is the statement label of an executable statement in the same scoping unit as the assigned **GO TO**. The same statement label can appear more than once in *stmt_label_list*.

Rules

When the assigned **GO TO** statement is executed, the variable you specify by *variable_name* with the value of a statement label must be defined. You must establish this definition with an **ASSIGN** statement in the same scoping unit as the assigned **GO TO** statement. If the integer variable is a dummy argument in a subprogram, you must assign it a statement label in the subprogram in order to use it in an assigned **GO TO** in that subprogram. Execution of the assigned **GO TO** statement transfers control to the statement identified by that statement label.

If *stmt_label_list* is present, the statement label assigned to the variable specified by *variable_name* must be one of the statement labels in the list.

The assigned **GO TO** cannot be the terminal statement of a **DO** or **DO WHILE** construct.

The assigned **GO TO** statement has been deleted in Fortran 95.

Examples

```
      INTEGER RETURN_LABEL
   ⋮
!  Simulate a call to a local procedure
      ASSIGN 100 TO RETURN_LABEL
      GOTO 9000
100   CONTINUE
```

```
   ⋮
9000  CONTINUE
!  A "local" procedure
   ⋮
      GOTO RETURN_LABEL
```

Related Information

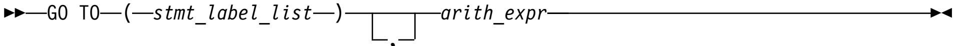
- "GO TO (Assigned)" on page 324
- "Statement Labels" on page 15
- "Branching" on page 131
- "Deleted Features" on page 735

GO TO (Computed)

Purpose

The computed **GO TO** statement transfers program control to one of possibly several executable statements.

Format



stmt_label
: is the statement label of an executable statement in the same scoping unit as the computed **GO TO**. The same statement label can appear more than once in *stmt_label_list*.

arith_expr
: is a scalar integer, real, or complex expression. If the value of the expression is noninteger, XL Fortran converts it to **INTEGER(4)** before using it.

Rules

When a computed **GO TO** statement is executed, the *arith_expr* is evaluated. The resulting value is used as an index into *stmt_label_list*. Control then transfers to the statement whose statement label you identify by the index. For example, if the value of *arith_expr* is 4, control transfers to the statement whose statement label is fourth in the *stmt_label_list*, provided there are at least four labels in the list.

If the value of *arith_expr* is less than 1 or greater than the number of statement labels in the list, the **GO TO** statement has no effect (like a **CONTINUE** statement), and the next statement is executed.

Examples

```
      INTEGER NEXT
    ⋮
      GO TO (100,200) NEXT
10    PRINT *,'Control transfers here if NEXT does not equal 1 or 2'
    ⋮
100   PRINT *,'Control transfers here if NEXT = 1'
    ⋮
200   PRINT *,'Control transfers here if NEXT = 2'
```

Related Information

- "Statement Labels" on page 15
- "Branching" on page 131

GO TO (Unconditional)

Purpose

The unconditional **GO TO** statement transfers program control to a specified executable statement.

Format

```
►►──GO TO──stmt_label──────────────────────────────────────────────►◄
```

stmt_label
: is the statement label of an executable statement in the same scoping unit as the unconditional **GO TO**

Rules

The unconditional **GO TO** statement transfers control to the statement identified by *stmt_label*.

The unconditional **GO TO** statement cannot be the terminal statement of a **DO** or **DO WHILE** construct.

Examples

```
   REAL(8) :: X,Y
   GO TO 10
   .
   .
   .
10 PRINT *, X,Y
   END
```

Related Information

- "Statement Labels" on page 15
- "Branching" on page 131

IF (Arithmetic)

Purpose

The arithmetic **IF** statement transfers program control to one of three executable statements, depending on the evaluation of an arithmetic expression.

Format

```
►►—IF—(—arith_expr—)—stmt_label1—,—stmt_label2—,—stmt_label3——————————►◄
```

arith_expr
: is a scalar arithmetic expression of type integer or real

stmt_label1, *stmt_label2*, and *stmt_label3*
: are statement labels of executable statements within the same scoping unit as the **IF** statement. The same statement label can appear more than once among the three statement labels.

Rules

The arithmetic **IF** statement evaluates *arith_expr* and transfers control to the statement identified by *stmt_label1*, *stmt_label2*, *or stmt_label3*, depending on whether the value of *arith_expr* is less than zero, zero, or greater than zero, respectively.

Examples

```
      IF (K-100) 10,20,30
10    PRINT *,'K is less than 100.'
      GO TO 40
20    PRINT *,'K equals 100.'
      GO TO 40
30    PRINT *,'K is greater than 100.'
40    CONTINUE
```

Related Information

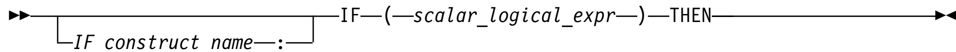
- "Branching" on page 131
- "Statement Labels" on page 15

IF (Block)

Purpose

The block **IF** statement is the first statement in an **IF** construct.

Format



IF_construct_name
: Is a name that identifies the **IF** construct.

Rules

The block **IF** statement evaluates a logical expression and executes at most one of the blocks contained within the **IF** construct.

If the *IF_construct_name* is specified, it must appear on the **END IF** statement, and optionally on any **ELSE IF** or **ELSE** statements in the **IF** construct.

Examples

```
WHICHC: IF (CMD .EQ. 'RETRY') THEN
     IF (LIMIT .GT. FIVE) THEN         ! Nested IF constructs
   ⋮
         CALL STOP
     ELSE
         CALL RETRY
     END IF
ELSE IF (CMD .EQ. 'STOP') THEN WHICHC
     CALL STOP
ELSE IF (CMD .EQ. 'ABORT') THEN
     CALL ABORT
ELSE WHICHC
     GO TO 100
END IF WHICHC
```

Related Information

- "IF Construct" on page 121
- "ELSE IF" on page 294
- "ELSE" on page 294
- "END (Construct)" on page 298, for details on the **END IF** statement

IF (Logical)

Purpose

The logical **IF** statement evaluates a logical expression and, if true, executes a specified statement.

Format

```
▸▸—IF—(—logical_expr—)—stmt——————————————————————————————————▸◂
```

logical_expr
is a scalar logical expression

stmt is an unlabeled executable statement

Rules

When a logical **IF** statement is executed, the *logical_expr* is evaluated. If the value of *logical_expr* is true, *stmt* is executed. If the value of *logical_expr* is false, *stmt* does not execute and the **IF** statement has no effect (like a **CONTINUE** statement).

Execution of a function reference in *logical_expr* can change the values of variables that appear in *stmt*.

stmt cannot be a **SELECT CASE, CASE, END SELECT, DO**, **DO WHILE**, **END DO**, block **IF**, **ELSE IF**, **ELSE**, **END IF**, **END FORALL**, another logical **IF**, **ELSEWHERE**, **END WHERE**, **END**, **END FUNCTION**, **END SUBROUTINE** statement, **FORALL** construct statement or **WHERE** construct statement.

Examples

```
IF (ERR.NE.0) CALL ERROR(ERR)
```

Related Information

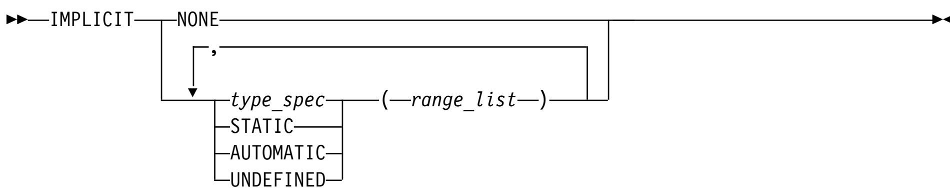
"Chapter 6. Control" on page 121

IMPLICIT

Purpose

The **IMPLICIT** statement changes or confirms the default implicit typing or the default storage class for local entities or, with the form **IMPLICIT NONE** specified, voids the implicit type rules altogether.

Format



type_spec
: specifies a data type. See "Type Declaration" on page 402.

range
: is either a single letter or range of letters. A range of letters has the form *letter*$_1$-*letter*$_2$, where *letter*$_1$ is the first letter in the range and *letter*$_2$, which follows *letter*$_1$ alphabetically, is the last letter in the range. Dollar sign ($) and underscore (_) are also permitted in a range. The underscore (_) follows the dollar sign ($), which follows the Z. Thus, the range Y - _ is the same as Y, Z, $, _.

Rules

Letter ranges cannot overlap; that is, no more than one type can be specified for a given letter.

In a given scoping unit, if a character has not been specified in an **IMPLICIT** statement, the implicit type for entities in a program unit or interface body is default integer for entities that begin with the characters I-N, and default real otherwise. The default for an internal or module procedure is the same as the implicit type used by the host scoping unit.

For any data entity name that begins with the character specified by *range_list*, and for which you do not explicitly specify a type, the type specified by the immediately preceding *type_spec* is provided. Note that implicit typing can be to a derived type that is inaccessible in the local scope if the derived type is accessible to the host scope.

A character or a range of characters that you specify as **STATIC** or **AUTOMATIC** can also appear in an **IMPLICIT** statement for any data type. A letter in a *range_list* cannot have both *type_spec* and **UNDEFINED** specified for it in the scoping unit. Neither can both **STATIC** and **AUTOMATIC** be specified for the same letter.

If you specify the form **IMPLICIT NONE** in a scoping unit, you must use type declaration statements to specify data types for names local to that scoping unit. You cannot refer to a name that does not have an explicitly defined data type; this lets you control all names that are inadvertently referenced. When **IMPLICIT NONE** is specified, you cannot specify any other

IMPLICIT statement in the same scoping unit, except ones that contain **STATIC** or **AUTOMATIC**. You can compile your program with the **-qundef** compiler option to achieve the same effect as an **IMPLICIT NONE** statement appearing in each scoping unit where an **IMPLICIT** statement is allowed.

IMPLICIT UNDEFINED turns off the implicit data typing defaults for the character or range of characters specified. When you specify **IMPLICIT UNDEFINED**, you must declare the data types of all symbolic names in the scoping unit that start with a specified character. The compiler issues a diagnostic message for each symbolic name local to the scoping unit that does not have an explicitly defined data type.

An **IMPLICIT** statement does not change the data type of an intrinsic function.

Using the **-qsave**/**-qnosave** compiler option modifies the predefined conventions for storage class:

-qsave compiler option	makes the predefined convention	IMPLICIT STATIC(a - _)
-qnosave compiler option	makes the predefined convention	IMPLICIT AUTOMATIC(a - _)

Even if you specified the **-qmixed** compiler option, the range list items are not case sensitive. For example, with **-qmixed** specified, `IMPLICIT INTEGER(A)` affects the implicit typing of data objects that begin with `A` as well as those that begin with `a`.

Examples

```
      IMPLICIT INTEGER (B), COMPLEX (D, K-M), REAL (R-Z,A)
!  This IMPLICIT statement establishes the following
!  implicit typing:
!
!        A: real
!        B: integer
!        C: real
!        D: complex
!   E to H: real
!     I, J: integer
!  K, L, M: complex
!        N: integer
!   O to Z: real
!        $: real
!        _: real
```

Related Information

- “How Type Is Determined” on page 52 for a discussion of the implicit rules
- “Storage Classes for Variables” on page 59
- ″**-qundef** Option″ in the *User's Guide*

- "**-qsave** Option" in the *User's Guide*

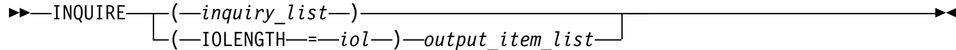
INQUIRE

Purpose

The **INQUIRE** statement obtains information about the properties of a named file or the connection to a particular unit.

There are three forms of the **INQUIRE** statement:
- Inquire by file, which requires the **FILE=** specifier to be specified.
- Inquire by unit, which requires the **UNIT=** specifier to be specified.
- Inquire by output list, which requires only the **IOLENGTH=** specifier to be specified.

Format



iol indicates the number of bytes of data that would result from the use of the output list in an unformatted output statement. *iol* is a scalar integer variable.

output_item
: See the **PRINT** or **WRITE** statement

inquiry_list
: is a list of inquiry specifiers for the inquire-by-file and inquire-by-unit forms of the **INQUIRE** statement. The inquire-by-file form cannot contain a unit specifier, and the inquire-by-unit form cannot contain a file specifier. No specifier can appear more than once in any **INQUIRE** statement. The inquiry specifiers are:

[UNIT=] *u*
: is a unit specifier. It specifies the unit about which the inquire-by-unit form of the statement is inquiring. *u* must be an external unit identifier whose value is not an asterisk. An external unit identifier refers to an external file that is represented by a scalar integer expression, whose value is in the range 0 through 2147483647. If the optional characters **UNIT=** are omitted, *u* must be the first item in *inquiry_list*.

IOSTAT= *ios*
: is an input/output status specifier that specifies the status of the input/output operation. *ios* is a scalar variable of type **INTEGER(4)** or default integer. When the input/output statement containing this specifier is finished executing, *ios* is defined with:

- A zero value if no error condition occurs
- A positive value if an error occurs.

Coding the **IOSTAT=** specifier suppresses error messages.

ERR= *stmt_label*
is an error specifier that specifies the statement label of an executable statement in the same scoping unit to which control is to transfer in the case of an error. Coding the **ERR=** specifier suppresses error messages.

FILE= *char_expr*
is a file specifier. It specifies the name of the file about which the inquire-by-file form of the statement is inquiring. *char_expr* is a scalar character expression whose value, when any trailing blanks are removed, is a valid AIX operating system file name. The named file does not have to exist, nor does it have to be associated with a unit.

Note: A valid AIX operating system file name must have a full path name of total length ≤ 1023 characters, with each file name ≤ 255 characters long (though the full path name need not be specified).

ACCESS= *char_var*
indicates whether the file is connected for sequential access or direct access. *char_var* is a scalar character variable that is assigned the value **SEQUENTIAL** if the file is connected for sequential access. The value assigned is **DIRECT** if the file is connected for direct access. If there is no connection, *char_var* is assigned the value **UNDEFINED**.

FORM= *char_var*
indicates whether the file is connected for formatted or unformatted input/output. *char_var* is a scalar default character variable that is assigned the value **FORMATTED** if the file is connected for formatted input/output. The value assigned is **UNFORMATTED** if the file is connected for unformatted input/output. If there is no connection, *char_var* is assigned the value **UNDEFINED**.

ASYNCH= *char_variable*
indicates whether the unit is connected for asynchronous access.

char_variable is a character variable that returns the value:
- **YES** if the unit is connected for both synchronous and asynchronous access;
- **NO** if the unit is connected for synchronous access only; or
- **UNDEFINED** if the unit is not connected.

TRANSFER= *char_variable*
is an asynchronous I/O specifier that indicates whether synchronous and/or asynchronous data transfer are permissible transfer methods for the file.

char_variable is a scalar character variable. If *char_variable* is assigned the value **BOTH**, then both synchronous and asynchronous data transfer are permitted. If *char_variable* is assigned the value **SYNCH**, then only synchronous data transfer is permitted. If *char_variable* is assigned the value **UNKNOWN**, then the processor is unable to determine the permissible transfer methods for this file.

RECL= *rcl*
indicates the value of the record length of a file connected for direct access, or the value of the maximum record length of a file connected for sequential access. *rcl* is a scalar variable of type **INTEGER(4)**, type **INTEGER(8)** in 64-bit, or type default integer that is assigned the value of the record length. If the file is connected for formatted input/output, the length is the number of characters for all records that contain character data. If the file is connected for unformatted input/output, the length is the number of bytes of data. If there is no connection, *rcl* becomes undefined.

BLANK= *char_var*
indicates the default treatment of blanks for a file connected for formatted input/output. *char_var* is a scalar character variable that is assigned the value **NULL** if all blanks in numeric input fields are ignored, or the value **ZERO** if all nonleading blanks are interpreted as zeros. If there is no connection, or if the connection is not for formatted input/output, *char_var* is assigned the value **UNDEFINED**.

EXIST= *ex*
indicates if a file or unit exists. *ex* is a scalar variable of type **LOGICAL(4)** or default logical that is assigned the value `true` or `false`. For the inquire-by-file form of the statement, the value `true` is assigned if the file specified by the **FILE=** specifier exists. The value `false` is assigned if the file does not exist. For the inquire-by-unit form of the statement, the value `true` is assigned if the unit specified by **UNIT=** exists. The value `false` is assigned if it is an invalid unit.

OPENED= *od*
indicates if a file or unit is connected. *od* is a scalar variable of type **LOGICAL(4)** or default logical that is assigned the value `true` or `false`. For the inquire-by-file form of the statement, the value `true` is assigned if the file specified by **FILE=** *char_var* is connected to a unit. The value `false` is assigned if the file is not connected to a unit. For the inquire-by-unit form of the statement, the value `true` is assigned if the unit specified by **UNIT=** is connected to a file. The value `false` is

assigned if the unit is not connected to a file. For preconnected files that have not been closed, the value is `true` both before and after the first input/output operation.

NUMBER= *num*
: indicates the external unit identifier currently associated with the file. *num* is a scalar variable of type **INTEGER(4)** or default integer that is assigned the value of the external unit identifier of the unit that is currently connected to the file. If there is no unit connected to the file, *num* is assigned the value `-1`.

NAMED= *nmd*
: indicates if the file has a name. *nmd* is a scalar variable of type **LOGICAL(4)** or default logical that is assigned the value `true` if the file has a name. The value assigned is `false` if the file does not have a name.

NAME= *fn*
: indicates the name of the file. *fn* is a scalar character variable that is assigned the name of the file to which the unit is connected.

SEQUENTIAL= *seq*
: indicates if the file is connected for sequential access. *seq* is a scalar character variable that is assigned the value **YES** if the file can be accessed sequentially, the value **NO** if the file cannot be accessed sequentially, or the value **UNKNOWN** if access cannot be determined.

DIRECT= *dir*
: indicates if the file is connected for direct access. *dir* is a scalar character variable that is assigned the value **YES** if the file can be accessed directly, the value **NO** if the file cannot be accessed directly, or the value **UNKNOWN** if access cannot be determined.

FORMATTED= *fmt*
: indicates if the file can be connected for formatted input/output. *fmt* is a scalar character variable that is assigned the value **YES** if the file can be connected for formatted input/output, the value **NO** if the file cannot be connected for formatted input/output, or the value **UNKNOWN** if formatting cannot be determined.

UNFORMATTED= *unf*
: indicates if the file can be connected for unformatted input/output. *fmt* is a scalar character variable that is assigned the value **YES** if the file can be connected for unformatted input/output, the value **NO** if the file cannot be connected for unformatted input/output, or the value **UNKNOWN** if formatting cannot be determined.

NEXTREC= *nr*
: indicates where the next record can be read or written on a file connected for direct access. *nr* is a scalar variable of type

INTEGER(4), **INTEGER(8)**, or default integer that is assigned the value `n + 1`, where `n` is the record number of the last record read or written on the file connected for direct access. If the file is connected but no records were read or written since the connection, *nr* is assigned the value 1. If the file is not connected for direct access or if the position of the file cannot be determined because of a previous error, *nr* becomes undefined.

Because record numbers can be greater than 2**31-1, you may choose to make the scalar variable specified with the **NEXTREC=** specifier of type **INTEGER(8)**. This could be accomplished in many ways, two examples include:
- Explicitly declaring *nr* as **INTEGER(8)**
- Changing the default kind of integers with the *-qintsize=8* compiler option.

POSITION= *pos*
indicates the position of the file. *pos* is a scalar character variable that is assigned the value **REWIND** if the file is connected by an **OPEN** statement for positioning at its initial point, **APPEND** if the file is connected for positioning before its endfile record or at its terminal point, **ASIS** if the file is connected without changing its position, or **UNDEFINED** if there is no connection or if the file is connected for direct access.

If the file has been repositioned to its initial point since it was opened, *pos* is assigned the value **REWIND**. If the file has been repositioned just before its endfile record since it was opened (or, if there is no endfile record, at its terminal point), *pos* is assigned the value **APPEND**. If both of the above are true and the file is empty, *pos* is assigned the value **APPEND**. If the file is positioned after the endfile record, *pos* is assigned the value **ASIS**.

ACTION= *act*
indicates if the file is connected for read and/or write access. *act* is a scalar character variable that is assigned the value **READ** if the file is connected for input only, **WRITE** if the file is connected for output only, **READWRITE** if the file is connected for both input and output, and **UNDEFINED** if there is no connection.

READ= *rd*
indicates if the file can be read. *rd* is a scalar character variable that is assigned the value **YES** if the file can be read, **NO** if the file cannot be read, and **UNKNOWN** if it cannot be determined if the file can be read.

WRITE= *wrt*
indicates if the file can be written to. *wrt* is a scalar character variable

that is assigned the value **YES** if the file can be written to, **NO** if the file cannot be written to, and **UNKNOWN** if it cannot be determined if the file can be written to.

READWRITE= *rw*
: indicates if the file can be both read from and written to. *rw* is a scalar character variable that is assigned the value **YES** if the file can be both read from and written to, **NO** if the file cannot be both read from and written to, and **UNKNOWN** if it cannot be determined if the file can be both read from and written to.

DELIM= *del*
: indicates the form, if any, that is used to delimit character data that is written by list-directed or namelist formatting. *del* is a scalar character variable that is assigned the value **APOSTROPHE** if apostrophes are used to delimit data, **QUOTE** if quotation marks are used to delimit data, **NONE** if neither apostrophes nor quotation marks are used to delimit data, and **UNDEFINED** if there is no file connection or no connection to formatted data.

PAD= *pd*
: indicates if the connection of the file had specified **PAD=NO**. *pd* is a scalar character variable that is assigned the value **NO** if the connection of the file had specified **PAD=NO**, and **YES** for all other cases.

Rules

An **INQUIRE** statement can be executed before, while, or after a file is associated with a unit. Any values assigned as the result of an **INQUIRE** statement are values that are current at the time the statement is executed.

If the unit or file is connected, the values returned for the **ACCESS=**, **SEQUENTIAL=**, **DIRECT=**, **ACTION=**, **READ=**, **WRITE=**, **READWRITE=**, **FORM=**, **FORMATTED=**, **UNFORMATTED=**, **BLANK=**, **DELIM=**, **PAD=**, **RECL=**, **POSITION=**, **NEXTREC=**, **NUMBER=**, **NAME=** and **NAMED=** specifiers are properties of the connection, and not of that file. Note that the **EXIST=** and **OPENED=** specifiers return `true` in these situations.

If a unit or file is not connected or does not exist, the **ACCESS=**, **ACTION=**, **FORM=**, **BLANK=**, **DELIM=**, **POSITION=** specifiers return the value **UNDEFINED**, the **DIRECT=**, **SEQUENTIAL=**, **FORMATTED=**, **UNFORMATTED=**, **READ=**, **WRITE=** and **READWRITE=** specifiers return the value **UNKNOWN**, the **RECL=** and **NEXTREC=** specifier variables are not defined, the **PAD=** specifier returns the value **YES**, and the **OPENED** specifier returns the value `false`.

If a unit or file does not exist, the **EXIST=** and **NAMED=** specifiers return the value `false`, the **NUMBER=** specifier returns the value -1, and the **NAME=** specifier variable is not defined.

If a unit or file exists but is not connected, the **EXIST=** specifier returns the value `true`. For the inquire-by-unit form of the statement, the **NAMED=** specifier returns the value `false`, the **NUMBER=** specifier returns the unit number, and the **NAME=** specifier variable is undefined. For the inquire-by-file form of the statement, the **NAMED=** specifier returns the value `true`, the **NUMBER=** specifier returns -1, and the **NAME=** specifier returns the file name.

The same variable name must not be specified for more than one specifier in the same **INQUIRE** statement, and must not be associated with any other variable in the list of specifiers.

Examples

```
SUBROUTINE SUB(N)
  CHARACTER(N) A(5)
  INQUIRE (IOLENGTH=IOL) A(1)  ! Inquire by output list
  OPEN (7,RECL=IOL)
   :
END SUBROUTINE
```

Related Information

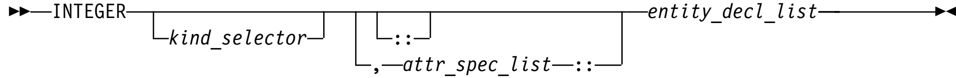
- "Conditions and IOSTAT Values" on page 193
- "Chapter 8. Input/Output Concepts" on page 183

INTEGER

Purpose

An **INTEGER** type declaration statement specifies the length and attributes of objects and functions of type integer. Initial values can be assigned to objects.

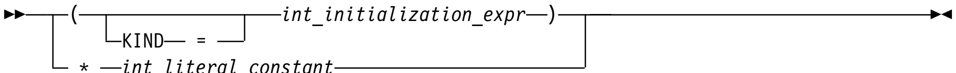
Format



where:

attr_spec
ALLOCATABLE **AUTOMATIC** **DIMENSION** (*array_spec*) **EXTERNAL** **INTENT** (*intent_spec*) **INTRINSIC** **OPTIONAL** **PARAMETER** **POINTER** **PRIVATE** **PUBLIC** **SAVE** **STATIC** **TARGET** **VOLATILE**

kind_selector



specifies the length of integer entities: 1, 2, 4 or 8. *int_literal_constant* cannot specify a kind type parameter.

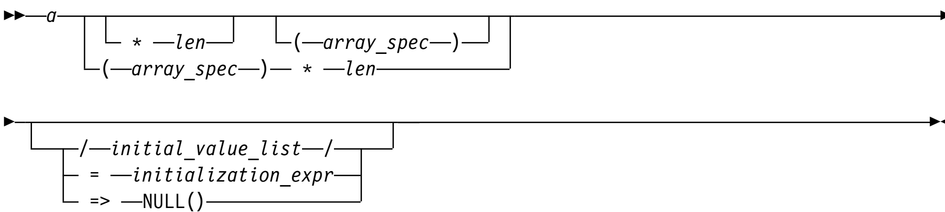
attr_spec
For detailed information on rules about a particular attribute, refer to the statement of the same name.

intent_spec
is either **IN**, **OUT**, or **INOUT**

:: is the double colon separator. It is required if attributes are specified, = *initialization_expr* or => **NULL()** is used.

array_spec
is a list of dimension bounds

entity_decl



a is an object name or function name. *array_spec* cannot be specified for a function name.

len overrides the length as specified in *kind_selector*, and cannot specify a kind type parameter. The entity length must be an integer literal constant that represents one of the permissible length specifications.

initial_value
provides an initial value for the entity specified by the immediately preceding name

initialization_expr
provides an initial value, by means of an initialization expression, for the entity specified by the immediately preceding name

=> **NULL()**
provides the initial value for the pointer object

Rules

Within the context of a derived type definition:

- If => appears in a component initialization, the **POINTER** attribute must appear in the *attr_spec_list*.
- If = appears in a component initialization, the **POINTER** attribute cannot appear in the component *attr_spec_list*.
- The compiler will evaluate *initialization_expr* within the scoping unit of the type definition.

If => appears for a variable, the object must have the **POINTER** attribute.

If *initialization_expr* appears for a variable, the object cannot have the **POINTER** attribute.

Entities in type declaration statements are constrained by the rules of any attributes specified for the entities, as detailed in the corresponding attribute statements.

The type declaration statement overrides the implicit type rules in effect. You can use a type declaration statement that confirms the type of an intrinsic function. The appearance of a generic or specific intrinsic function name in a type declaration statement does not cause the name to lose its intrinsic property.

An object cannot be initialized in a type declaration statement if it is a dummy argument, an allocatable array, a pointer, a function result, an object in blank common, an integer pointer, an external name, an intrinsic name, or an automatic object. Nor can an object be initialized if it has the **AUTOMATIC** attribute. The object may be initialized if it appears in a named common block in a block data program unit or if it appears in a named common block in a module.

In Fortran 95, a pointer can be initialized. Pointers can only be initialized by the use of **=> NULL()**.

The specification expression of an *array_spec* can be a nonconstant expression if the specification expression appears in an interface body or in the specification part of a subprogram. Any object being declared that uses this nonconstant expression and is not a dummy argument or a pointee is called an *automatic object*.

An attribute cannot be repeated in a given type declaration statement, nor can an entity be explicitly given the same attribute more than once in a scoping unit.

initialization_expr must be specified if the statement contains the **PARAMETER** attribute. If the entity you are declaring is a variable, and *initialization_expr* or **NULL()** is specified, the variable is initially defined. If the entity you are declaring is a derived type component, and *initialization_expr* or **NULL()** is specified, the derived type has default initialization. *a* becomes defined with the value determined by *initialization_expr*, in accordance with the rules for intrinsic assignment. If the entity is an array, its shape must be specified either in the type declaration statement or in a previous specification statement in the same scoping unit. A variable or variable subobject cannot be initialized more than once. If *a* is a variable, the presence of *initialization_expr* or => **NULL()** implies that *a* is a saved object, except for an object in a named common block. The initialization of an object could affect the fundamental storage class of an object.

An *array_spec* specified in the *entity_decl* takes precedence over the *array_spec* in the **DIMENSION** attribute.

An array function result that does not have the **POINTER** attribute must have an explicit-shape array specification.

If the entity declared is a function, it must not have an accessible explicit interface unless it is an intrinsic function.

If T or F, defined previously as the name of a constant, appears in a type declaration statement, it is no longer an abbreviated logical constant but the name of the named constant.

Examples

```
MODULE INT
  INTEGER, DIMENSION(3) :: A,B,C
  INTEGER :: X=234,Y=678
END MODULE INT
```

Related Information

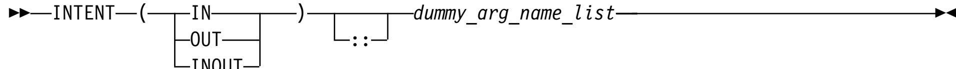
- "Integer" on page 29
- "Initialization Expressions" on page 87
- "How Type Is Determined" on page 52, for details on the implicit typing rules
- "Array Declarators" on page 65
- "Automatic Objects" on page 29
- "Storage Classes for Variables" on page 59
- "DATA" on page 277, for details on initial values

INTENT

Purpose

The **INTENT** attribute specifies the intended use of dummy arguments.

Format



dummy_arg_name
: is the name of a dummy argument, which cannot be a dummy procedure or a dummy pointer

Rules

The **INTENT** attribute can take three forms:

- **INTENT(IN)** specifies that the dummy argument must not be redefined or become undefined during the execution of the subprogram.
- **INTENT(OUT)** specifies that the dummy argument must be defined before it is referenced within the subprogram. Such a dummy argument might not become undefined on invocation of the subprogram.
- **INTENT(INOUT)** specifies that the dummy argument can both receive and return data to the invoking subprogram.

An actual argument that becomes associated with a dummy argument with an intent of **OUT** or **INOUT** must be definable. Hence, a dummy argument with an intent of **IN**, or an actual argument that is a constant, a subobject of a constant, or an expression, cannot be passed as an actual argument to a subprogram expecting an argument with an intent of **OUT** or **INOUT**.

An actual argument that is an array section with a vector subscript cannot be associated with a dummy array that is defined or redefined (that is, with an intent of **OUT** or **INOUT**).

The **%VAL** built-in function, used for interlanguage calls, can only be used for an actual argument that corresponds to a dummy argument with an intent of **IN**, or that has no intent specified. This constraint does not apply to the **%REF** built-in function.

Attributes Compatible with the INTENT Attribute

- DIMENSION
- OPTIONAL
- TARGET
- VOLATILE

Examples

```
PROGRAM MAIN
  DATA R,S /12.34,56.78/
  CALL SUB(R+S,R,S)
END PROGRAM

SUBROUTINE SUB (A,B,C)
  INTENT(IN) A
  INTENT(OUT) B
  INTENT(INOUT) C
  C=C+A+ABS(A)              ! Valid references to A and C
                            ! Valid redefinition of C
  B=C**2                    ! Valid redefinition of B
END SUBROUTINE
```

Related Information

- “Intent of Dummy Arguments” on page 166
- “Argument Association” on page 163
- “%VAL and %REF” on page 165, for details on interlanguage calls

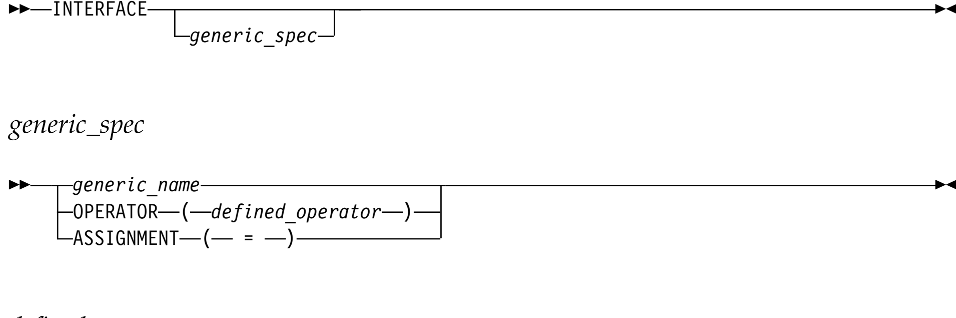
- “Dummy Arguments” on page 162

INTERFACE

Purpose

The **INTERFACE** statement is the first statement of an interface block, which can specify an explicit interface for an external or dummy procedure.

Format



defined_operator
: is a defined unary operator, defined binary operator, or extended intrinsic operator

Rules

If *generic_spec* is present, the interface block is generic. If *generic_spec* is absent, the interface block is nongeneric. *generic_name* specifies a single name to reference all procedures in the interface block. At most, one specific procedure is invoked each time there is a procedure reference with a generic name.

If a *generic_spec* appears in an **INTERFACE** statement, it must match the *generic_spec* in the corresponding **END INTERFACE** statement.

An **INTERFACE** statement without a *generic_spec* can match any **END INTERFACE** statement, with or without a *generic_spec*.

If the *generic_spec* in an **INTERFACE** statement is a *generic_name*, the *generic_spec* of the corresponding **END INTERFACE** statement must be the same *generic_name*.

A specific procedure must not have more than one explicit interface in a given scoping unit.

You can always reference a procedure through its specific interface, if accessible. If a generic interface exists for a procedure, the procedure can also be referenced through the generic interface.

If *generic_spec* is **OPERATOR**(*defined_operator*), the interface block can define a defined operator or extend an intrinsic operator.

If *generic_spec* is **ASSIGNMENT**(=), the interface block can extend intrinsic assignment.

Examples

```
INTERFACE                             ! Nongeneric interface block
  FUNCTION VOL(RDS,HGT)
    REAL VOL, RDS, HGT
  END FUNCTION VOL
  FUNCTION AREA (RDS)
    REAL AREA, RDS
  END FUNCTION AREA
END INTERFACE

INTERFACE OPERATOR (.DETERMINANT.)    ! Defined operator interface
  FUNCTION DETERMINANT(X)
    INTENT(IN) X
    REAL X(50,50), DETERMINANT
  END FUNCTION
END INTERFACE

INTERFACE ASSIGNMENT(=)               ! Defined assignment interface
  SUBROUTINE BIT_TO_NUMERIC (N,B)
    INTEGER, INTENT(OUT) :: N
    LOGICAL, INTENT(IN)  :: B(:)
  END SUBROUTINE
END INTERFACE
```

Related Information

- "Explicit Interface" on page 144
- "Extended Intrinsic and Defined Operations" on page 99
- "Defined Operators" on page 150
- "Defined Assignment" on page 151
- "FUNCTION" on page 320
- "SUBROUTINE" on page 396
- "MODULE PROCEDURE" on page 351
- "Procedure References" on page 159
- "Unambiguous Generic Procedure References" on page 147, for details about the rules on how any two procedures with the same generic name must differ

---

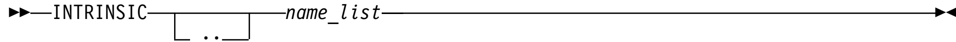
INTRINSIC

Purpose

The **INTRINSIC** attribute identifies a name as an intrinsic procedure and allows you to use specific names of intrinsic procedures as actual arguments.

INTRINSIC

Format



name is the name of an intrinsic procedure

Rules

If you use a specific intrinsic procedure name as an actual argument in a scoping unit, it must have the **INTRINSIC** attribute. Generic names can have the **INTRINSIC** attribute, but you cannot pass them as arguments unless they are also specific names.

A generic or specific procedure that has the **INTRINSIC** attribute keeps its generic or specific properties.

A generic intrinsic procedure that has the **INTRINSIC** attribute can also be the name of a generic interface block. The generic interface block defines extensions to the generic intrinsic procedure.

Attributes Compatible with the INTRINSIC Attribute

- PRIVATE
- PUBLIC

Examples

```
PROGRAM MAIN
  INTRINSIC SIN, ABS
  INTERFACE ABS
    LOGICAL FUNCTION MYABS(ARG)
      LOGICAL ARG
    END FUNCTION
  END INTERFACE

  LOGICAL LANS,LVAR
  REAL(8) DANS,DVAR
  DANS = ABS(DVAR)            ! Calls the DABS intrinsic procedure
  LANS = ABS(LVAR)            ! Calls the MYABS external procedure

! Pass intrinsic procedure name to subroutine
  CALL DOIT(0.5,SIN,X)        ! Passes the SIN specific intrinsic
END PROGRAM

SUBROUTINE DOIT(RIN,OPER,RESULT)
  INTRINSIC :: MATMUL
  INTRINSIC    COS
  RESULT = OPER(RIN)
END SUBROUTINE
```

Related Information

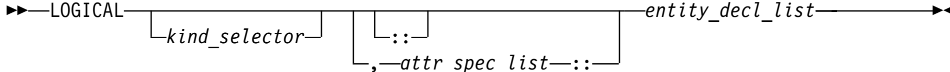
- Generic and specific intrinsic procedures are listed in “Chapter 12. Intrinsic Procedures” on page 519. See this section to find out if a specific intrinsic name can be used as an actual argument.
- “Generic Interface Blocks” on page 147

LOGICAL

Purpose

A **LOGICAL** type declaration statement specifies the length and attributes of objects and functions of type logical. Initial values can be assigned to objects.

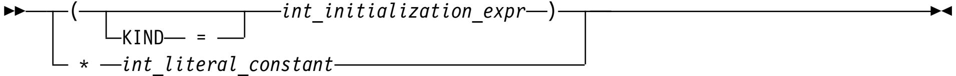
Format



where:

attr_spec
ALLOCATABLE
AUTOMATIC
DIMENSION (*array_spec*)
EXTERNAL
INTENT (*intent_spec*)
INTRINSIC
OPTIONAL
PARAMETER
POINTER
PRIVATE
PUBLIC
SAVE
STATIC
TARGET
VOLATILE

kind_selector



specifies the length of logical entities: 1, 2, 4 or 8. *int_literal_constant* cannot specify a kind type parameter.

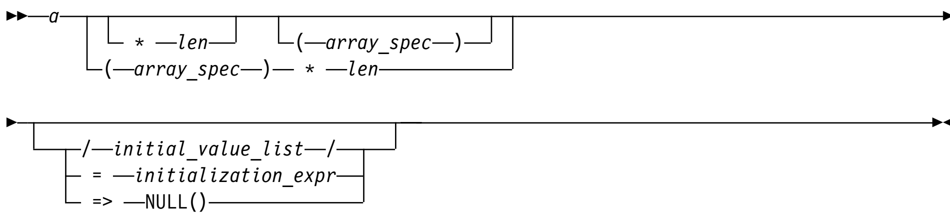
attr_spec
: For detailed information on rules about a particular attribute, refer to the statement of the same name.

intent_spec
: is either **IN**, **OUT**, or **INOUT**

::
: is the double colon separator. It is required if attributes are specified, = *initialization_expr* or => **NULL()** is used.

array_spec
: is a list of dimension bounds

entity_decl



a
: is an object name or function name. *array_spec* cannot be specified for a function name.

len
: overrides the length as specified in *kind_selector*, and cannot specify a kind type parameter. The entity length must be an integer literal constant that represents one of the permissible length specifications.

initial_value
: provides an initial value for the entity specified by the immediately preceding name

initialization_expr
: provides an initial value, by means of an initialization expression, for the entity specified by the immediately preceding name

=> **NULL()**
: provides the initial value for the pointer object

Rules

Within the context of a derived type definition:

- If => appears in a component initialization, the **POINTER** attribute must appear in the *attr_spec_list*.

- If = appears in a component initialization, the **POINTER** attribute cannot appear in the component *attr_spec_list*.
- The compiler will evaluate *initialization_expr* within the scoping unit of the type definition.

If => appears for a variable, the object must have the **POINTER** attribute.

If *initialization_expr* appears for a variable, the object cannot have the **POINTER** attribute.

Entities in type declaration statements are constrained by the rules of any attributes specified for the entities, as detailed in the corresponding attribute statements.

The type declaration statement overrides the implicit type rules in effect. You can use a type declaration statement that confirms the type of an intrinsic function. The appearance of a generic or specific intrinsic function name in a type declaration statement does not cause the name to lose its intrinsic property.

An object cannot be initialized in a type declaration statement if it is a dummy argument, an allocatable array, a pointer, a function result, an object in blank common, an integer pointer, an external name, an intrinsic name, or an automatic object. Nor can an object be initialized if it has the **AUTOMATIC** attribute. The object may be initialized if it appears in a named common block in a block data program unit or if it appears in a named common block in a module.

In Fortran 95, a pointer can be initialized. Pointers can only be initialized by the use of **=> NULL()**.

The specification expression of an *array_spec* can be a nonconstant expression if the specification expression appears in an interface body or in the specification part of a subprogram. Any object being declared that uses this nonconstant expression and is not a dummy argument or a pointee is called an *automatic object*.

An attribute cannot be repeated in a given type declaration statement, nor can an entity be explicitly given the same attribute more than once in a scoping unit.

initialization_expr must be specified if the statement contains the **PARAMETER** attribute. If the entity you are declaring is a variable, and *initialization_expr* or **NULL()** is specified, the variable is initially defined. If the entity you are declaring is a derived type component, and *initialization_expr* or **NULL()** is specified, the derived type has default initialization. *a* becomes defined with

the value determined by *initialization_expr*, in accordance with the rules for intrinsic assignment. If the entity is an array, its shape must be specified either in the type declaration statement or in a previous specification statement in the same scoping unit. A variable or variable subobject cannot be initialized more than once. If *a* is a variable, the presence of *initialization_expr* or => **NULL()** implies that *a* is a saved object, except for an object in a named common block. The initialization of an object could affect the fundamental storage class of an object.

An *array_spec* specified in the *entity_decl* takes precedence over the *array_spec* in the **DIMENSION** attribute.

An array function result that does not have the **POINTER** attribute must have an explicit-shape array specification.

If the entity declared is a function, it must not have an accessible explicit interface unless it is an intrinsic function.

If T or F, defined previously as the name of a constant, appears in a type declaration statement, it is no longer an abbreviated logical constant but the name of the named constant.

Examples

```
LOGICAL, ALLOCATABLE :: L(:,:)
LOGICAL :: Z=.TRUE.
```

Related Information

- “Logical” on page 34
- “Initialization Expressions” on page 87
- “How Type Is Determined” on page 52, for details on the implicit typing rules
- “Array Declarators” on page 65
- “Automatic Objects” on page 29
- “Storage Classes for Variables” on page 59
- “DATA” on page 277, for details on initial values

MODULE

Purpose

The **MODULE** statement is the first statement of a module program unit, which contains specifications and definitions that can be made accessible to other program units.

Format

```
►►—MODULE—module_name——————————————————————————————————►◄
```

Rules

The module name is a global entity that is referenced by the **USE** statement in other program units to access the public entities of the module. The module name must not have the same name as any other program unit, external procedure or common block in the program, nor can it be the same as any local name in the module.

If the **END** statement that completes the module specifies a module name, the name must be the same as that specified in the **MODULE** statement.

Examples

```
MODULE MM
   CONTAINS
     REAL FUNCTION SUM(CARG)
       COMPLEX CARG
       SUM_FNC(CARG) = IMAG(CARG) + REAL(CARG)
       SUM = SUM_FNC(CARG)
       RETURN
     ENTRY AVERAGE(CARG)
       AVERAGE = SUM_FNC(CARG) / 2.0
     END FUNCTION SUM
     SUBROUTINE SHOW_SUM(SARG)
       COMPLEX SARG
       REAL SUM_TMP
  10   FORMAT('SUM:',E10.3,' REAL:',E10.3,' IMAG',E10.3)
       SUM_TMP = SUM(CARG=SARG)
       WRITE(10,10) SUM_TMP, SARG
     END SUBROUTINE SHOW_SUM
END MODULE MM
```

Related Information

- “Modules” on page 153
- “USE” on page 408
- “Use Association” on page 139
- “END” on page 296, for details on the **END MODULE** statement
- “PRIVATE” on page 370
- “PUBLIC” on page 373

MODULE PROCEDURE

Purpose

The **MODULE PROCEDURE** statement lists those module procedures that have a generic interface.

MODULE PROCEDURE

Format

```
►►──MODULE PROCEDURE──procedure_name_list──────────────────────────►◄
```

Rules

The **MODULE PROCEDURE** statement can appear anywhere among the interface bodies in an interface block that has a generic specification.

MODULE PROCEDURE statements must be contained in a scoping unit where *procedure_name* can be accessed as a module procedure, and must be the name that is accesible in this scope.

procedure_name must not have been previously associated with the generic specification of the interface block in which it appears, either by a previous appearance in an interface block or by use or by host association.

The characteristics of module procedures are determined by module procedure definitions, not by interface bodies.

Examples

```
MODULE M
  CONTAINS
  SUBROUTINE S1(IARG)
    IARG=1
  END SUBROUTINE
  SUBROUTINE S2(RARG)
    RARG=1.1
  END SUBROUTINE
END MODULE

USE M
INTERFACE SS
  SUBROUTINE SS1(IARG,JARG)
  END SUBROUTINE
  MODULE PROCEDURE S1, S2
END INTERFACE
CALL SS(N)                    ! Calls subroutine S1 from M
CALL SS(I,J)                  ! Calls subroutine SS1
END
```

Related Information

- "Interface Blocks" on page 144
- "INTERFACE" on page 344
- "Modules" on page 153

NAMELIST

Purpose

The **NAMELIST** statement specifies one or more lists of names for use in **READ**, **WRITE**, and **PRINT** statements.

Format

Nname is a namelist group name

variable_name
must not be an array dummy argument with a nonconstant bound, a variable with nonconstant character length, an automatic object, a pointer, a variable of a type that has an ultimate component that is a pointer, an allocatable array, or a pointee.

Rules

The list of names belonging to a namelist group name ends with the appearance of another namelist group name or the end of the **NAMELIST** statement.

variable_name must either be accessed via use or host association, or have its type and type parameters specified by previous specification statements in the same scoping unit or by the implicit typing rules. If typed implicitly, any appearance of the object in a subsequent type declaration statement must confirm the implied type and type parameters. A derived-type object must not appear as a list item if any component ultimately contained within the object is not accessible within the scoping unit containing the namelist input/output statement on which its containing namelist group name is specified.

variable_name can belong to one or more namelist lists. If the namelist group name has the **PUBLIC** attribute, no item in the list can have the **PRIVATE** attribute or private components.

Nname can be specified in more than one **NAMELIST** statement in the scoping unit. The *variable_name_list* following each successive appearance of the same *Nname* in a scoping unit is treated as the continuation of the list for that *Nname*.

A namelist name can appear only in input/output statements. The rules for input/output conversion of namelist data are the same as the rules for data conversion.

Examples

```
DIMENSION X(5), Y(10)
NAMELIST /NAME1/ I,J,K
NAMELIST /NAME2/ A,B,C /NAME3/ X,Y
WRITE (10, NAME1)
PRINT NAME2
```

Related Information

- “Namelist Formatting” on page 231
- "Setting Runtime Options for Input/Output" in the *User's Guide*

NULLIFY

Purpose

The **NULLIFY** statement causes pointers to become disassociated.

Format

```
►►—NULLIFY—(—pointer_object_list—)——————————————————————————————►◄
```

pointer_object
: is a pointer variable name or structure component

Rules

A *pointer_object* must have the **POINTER** attribute.

> **Tip**
> Always initialize a pointer with the **NULLIFY** statement, pointer assignment, default initialization or explicit initialization.

Examples

```
TYPE T
  INTEGER CELL
  TYPE(T), POINTER :: NEXT
ENDTYPE T
TYPE(T) HEAD, TAIL
TARGET :: TAIL
HEAD%NEXT => TAIL
NULLIFY (TAIL%NEXT)
END
```

Related Information

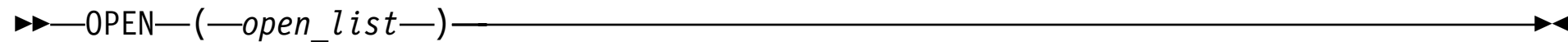
- “Pointer Assignment” on page 117
- “Pointer Association” on page 139

OPEN

Purpose

The **OPEN** statement can be used to connect an existing external file to a unit, create an external file that is preconnected, create an external file and connect it to a unit, or change certain specifiers of a connection between an external file and a unit.

Format



open_list
: is a list that must contain one unit specifier (**UNIT=***u*) and can also contain one of each of the other valid specifiers. The valid specifiers are:

[UNIT=] *u*
: is a unit specifier in which *u* must be an external unit identifier whose value is not an asterisk. An external unit identifier refers to an external file that is represented by a scalar integer expression, whose value is in the range 0 through 2,147,483,647. If the optional characters **UNIT=** are omitted, *u* must be the first item in *open_list*.

ASYNCH= *char_expr*
: is an asynchronous I/O specifier that indicates whether an explicitly connected unit is to be used for asynchronous I/O.

char_expr is a scalar character expression whose value is either **YES** or **NO**. **YES** specifies that asynchronous data transfer statements are permitted for this connection. **NO** specifies that asynchronous data transfer statements are not permitted for this connection. The value specified will be in the set of transfer methods permitted for the file. If this specifier is omitted, the default value is **NO**.

Preconnected units are connected with an **ASYNCH=** value of **NO**.

The **ASYNCH=** value of an implicitly connected unit is determined by the first data transfer statement performed on the unit. If the first statement performs an asynchronous data transfer and the file being implicitly connected permits asynchronous data transfers, the **ASYNCH=** value is **YES**. Otherwise, the **ASYNCH=** value is **NO**.

IOSTAT= *ios*
: is an input/output status specifier that specifies the status of the input/output operation. *ios* is a scalar variable of type **INTEGER(4)** or default integer. When the input/output statement containing this specifier finishes execution, *ios* is defined with:

- A zero value if no error condition occurs
- A positive value if an error occurs.

ERR= *stmt_label*
is an error specifier that specifies the statement label of an executable statement in the same scoping unit to which control is to transfer in the case of an error. Coding the **ERR=** specifier suppresses error messages.

FILE= *char_expr*
is a file specifier that specifies the name of the file to be connected to the specified unit.

char_expr is a scalar character expression whose value, when any trailing blanks are removed, is a valid AIX operating system file name. If the file specifier is omitted and is required, the unit becomes implicitly connected (by default) to **fort.***u*, where *u* is the unit specified with any leading zeros removed. Use the **UNIT_VARS** run-time option to allow alternative files names to be used for files that are implicitly connected.

Note: A valid AIX operating system file name must have a full path name of total length ≤1023 characters, with each file name ≤255 characters long (although the full path name need not be specified).

STATUS= *char_expr*
specifies the status of the file when it is opened. *char_expr* is a scalar character expression whose value, when any trailing blanks are removed, is one of the following:

- **OLD**, to connect an existing file to a unit. If **OLD** is specified, the file must exist. If the file does not exist, an error condition will occur.
- **NEW**, to create a new file, connect it to a unit, and change the status to **OLD**. If **NEW** is specified, the file must not exist. If the file already exists, an error condition will occur.
- **SCRATCH**, to create and connect a new file that will be deleted when it is disconnected. **SCRATCH** must not be specified with a named file (that is, **FILE=***char_expr* must be omitted).
- **REPLACE**. If the file does not already exist, the file is created and the status is changed to **OLD**. If the file exists, the file is deleted, a new file is created with the same name, and the status is changed to **OLD**.
- **UNKNOWN**, to connect an existing file, or to create and connect a new file. If the file exists, it is connected as **OLD**. If the file does not exist, it is connected as **NEW**.

UNKNOWN is the default.

ACCESS= *char_expr*
: specifies the access method for the connection of the file. *char_expr* is a scalar character expression whose value, when any trailing blanks are removed, is either **SEQUENTIAL** or **DIRECT**. **SEQUENTIAL** is the default. If **ACCESS** is **DIRECT**, **RECL=** must be specified. If **ACCESS** is **SEQUENTIAL**, **RECL=** is optional.

FORM= *char_expr*
: specifies whether the file is connected for formatted or unformatted input/output. *char_expr* is a scalar character expression whose value, when any trailing blanks are removed, is either **FORMATTED** or **UNFORMATTED**. If the file is being connected for sequential access, **FORMATTED** is the default. If the file is being connected for direct access, **UNFORMATTED** is the default.

RECL= *integer_expr*
: specifies the length of each record in a file being connected for direct access or the maximum length of a record in a file being connected for sequential access. *integer_expr* is a scalar integer expression whose value must be positive. This specifier must be present when a file is being connected for direct access. For formatted input/output, the length is the number of characters for all records that contain character data. For unformatted input/output, the length is the number of bytes required for the internal form of the data. The length of an unformatted sequential record does not count the four-byte fields surrounding the data.

If **RECL=** is omitted when a file is being connected for sequential access in 32-bit, the length is 2**31 minus the record terminator. For a formatted sequential file in 32-bit, the default record length is 2**31-1. For an unformatted file that can be accessed in 32-bit, the default record length is 2**31-8. For a file that cannot be accessed randomly in 32-bit, the default length is **32,768**.

If **RECL=** is omitted when a file is being connected for sequential access in 64-bit, the length is 2**64 minus the record terminator. For a formatted sequential file in 64-bit, the default record length is 2**64-1 For an unformatted file in 64-bit, the default record length is 2**64-16 when the **UWIDTH** run-time option is set to 64.

BLANK= *char_expr*
: controls the default interpretation of blanks when you are using a format specification. *char_expr* is a scalar character expression whose value, when any trailing blanks are removed, is either **NULL** or **ZERO**. If **BLANK=** is specified, you must use **FORM='FORMATTED'**. If **BLANK=** is not specified and you specify **FORM='FORMATTED'**, **NULL** is the default.

POSITION= *char_expr*
specifies the file position for a file connected for sequential access. A file that did not exist previously is positioned at its initial point. *char_expr* is a scalar character expression whose value, when any trailing blanks are removed, is either **ASIS**, **REWIND**, or **APPEND**. **REWIND** positions the file at its initial point. **APPEND** positions the file before the endfile record or, if there is no endfile record, at the terminal point. **ASIS** leaves the position unchanged. The default value is **ASIS** except under the following conditions:

- The first input/output statement (other than the **INQUIRE** statement) referring to the unit after the **OPEN** statement is a **WRITE** statement, and either:
  - The **STATUS=** specifier is **UNKNOWN** and the **-qposition** compiler option specifies **appendunknown**, or
  - The **STATUS=** specifier is **OLD** and the **-qposition** compiler option specifies **appendold**.

In such cases, the default value for the **POSITION=** specifier is **APPEND** at the time the **WRITE** statement is executed.

ACTION= *char_expr*
specifies the allowed input/output operations. *char_expr* is a scalar character expression whose value evaluates to **READ**, **WRITE** or **READWRITE**. If **READ** is specified, **WRITE** and **ENDFILE** statements cannot refer to this connection. If **WRITE** is specified, **READ** statements cannot refer to this connection. The value **READWRITE** permits any input/output statement to refer to this connection. If the **ACTION=** specifier is omitted, the default value depends on the actual file permissions:

- If the **STATUS=** specifier has the value **OLD** or **UNKNOWN** and the file already exists:
  - The file is opened with **READWRITE**
  - If the above is not possible, the file is opened with **READ**
  - If neither of the above is possible, the file is opened with **WRITE**.
- If the **STATUS=** specifier has the value **NEW**, **REPLACE**, **SCRATCH** or **UNKNOWN** and the file does not exist:
  - The file is opened with **READWRITE**
  - If the above is not possible, the file is opened with **WRITE**.

DELIM= *char_expr*
specifies what delimiter, if any, is used to delimit character constants written with list-directed or namelist formatting. *char_expr* is a scalar character expression whose value must evaluate to **APOSTROPHE**, **QUOTE**, or **NONE**. If the value is **APOSTROPHE**, apostrophes delimit character constants and all apostrophes within character constants are doubled. If the value is **QUOTE**, double quotation

marks delimit character constants and all double quotation marks within character constants are doubled. If the value is **NONE**, character constants are not delimited and no characters are doubled. The default value is **NONE**. The **DELIM=** specifier is permitted only for files being connected for formatted input/output, although it is ignored during input of a formatted record.

PAD= *char_expr*

specifies if input records are padded with blanks. *char_expr* is a scalar character expression that 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. If **NO** is specified, the input list and format specification must not require more characters from a record than the record contains. The default value is **YES**. The **PAD=** specifier is permitted only for files being connected for formatted input/output, although it is ignored during output of a formatted record.

If the **-qxlf77** compiler option specifies the **noblankpad** suboption and the file is being connected for formatted direct input/output, the default value is **NO** when the **PAD=** specifier is omitted.

Rules

If a unit is connected to a file that exists, an **OPEN** statement for that unit can be performed. If the **FILE=** specifier is not included in the **OPEN** statement, the file to be connected to the unit is the same as the file to which the unit is connected.

If the file to be connected to the unit is not the same as the file to which the unit is connected, the effect is as if a **CLOSE** statement without a **STATUS=** specifier had been executed for the unit immediately prior to the execution of the **OPEN** statement.

If the file to be connected to the unit is the same as the file to which the unit is connected, only the **BLANK=**, **DELIM=**, **PAD=**, **ERR=**, and **IOSTAT=** specifiers can have a value different from the one currently in effect. Execution of the **OPEN** statement causes any new value for the **BLANK=**, **DELIM=** or **PAD=** specifiers to be in effect, but does not cause any change in any of the unspecified specifiers or the position of the file. Any **ERR=** and **IOSTAT=** specifiers from **OPEN** statements previously executed have no effect on the current **OPEN** statement. If you specify the **STATUS=** specifier it must have the value **OLD**. To specify the same file as the one currently connected to the unit, you can specify the same file name, omit the **FILE=** specifier, or specify a file symbolically linked to the same file.

If a file is connected to a unit, an **OPEN** statement on that file and a different unit cannot be performed.

If the **STATUS=** specifier has the value **OLD**, **NEW** or **REPLACE**, the **FILE=** specifier is optional.

Unit 0 cannot be specified to connect to a file other than the preconnected file, the standard error device, although you can change the values for the **BLANK=**, **DELIM=** and **PAD=** specifiers.

If the **ERR=** and **IOSTAT=** specifiers are set and an error is encountered, transfer is made to the statement specified by the **ERR=** specifier and a positive integer value is assigned to *ios*.

If **IOSTAT=** and **ERR=** are not specified,
- The program stops if a severe error is encountered
- The program continues to the next statement if a recoverable error is encountered and the **ERR_RECOVERY** run-time option is set to **YES**. If the option is set to **NO**, the program stops.

Examples

```
!   Open a new file with name fname

CHARACTER*20 FNAME
FNAME = 'INPUT.DAT'
OPEN(UNIT=8,FILE=FNAME,STATUS='NEW',FORM='FORMATTED')

OPEN (4,FILE="myfile")
OPEN (4,FILE="myfile", PAD="NO")  ! Changing PAD= value to NO

!   Connects unit 2 to a tape device for unformatted, sequential
!   write-only access:

OPEN (2, FILE="/dev/rmt0",ACTION="WRITE",POSITION="REWIND", &
&   FORM="UNFORMATTED",ACCESS="SEQUENTIAL",RECL=32767)
```

Related Information

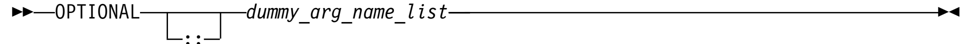
- “Connection of a Unit” on page 186
- Item 3 under “Appendix A. Compatibility Across Standards” on page 731
- “Conditions and IOSTAT Values” on page 193
- ″Setting Runtime Options for Input/Output″ in the *User′s Guide*
- ″-qposition Option″ in the *User′s Guide*
- ″-qxlf77 Option″ in the *User′s Guide*
- “CLOSE” on page 265
- “READ” on page 374
- “WRITE” on page 416

OPTIONAL

Purpose

The **OPTIONAL** attribute specifies that a dummy argument need not be associated with an actual argument in a reference to the procedure.

Format



Rules

A reference to a procedure that has an optional dummy argument specified must have an explicit interface.

Use the **PRESENT** intrinsic function to determine if an actual argument has been associated with an optional dummy argument. Avoid referencing an optional dummy argument without first verifying that the dummy argument is present.

A dummy argument is considered present in a subprogram if it is associated with an actual argument, which itself can also be a dummy argument that is present (an instance of propagation). A dummy argument that is not optional must be present; that is, it must be associated with an actual argument.

An optional dummy argument that is not present may be used as an actual argument corresponding to an optional dummy argument, which is then also considered not to be associated with an actual argument. An optional dummy argument that is not present is subject to the following restrictions:

- If it is a dummy data object or subobject, it cannot be defined or referenced.
- If it is a dummy procedure, it cannot be referenced.
- It cannot appear as an actual argument corresponding to a non-optional dummy argument, other than as the argument of the **PRESENT** intrinsic function.
- If it is an array, it must not be supplied as an actual argument to an elemental procedure unless an array of the same rank is supplied as an actual argument, which corresponds to a nonoptional argument of that elemental procedure.

The **OPTIONAL** attribute cannot be specified for dummy arguments in an interface body that specifies an explicit interface for a defined operator or defined assignment.

Attributes Compatible with the OPTIONAL Attribute

- DIMENSION
- EXTERNAL
- INTENT
- POINTER
- TARGET
- VOLATILE

Examples

```
SUBROUTINE SUB (X,Y)
  INTERFACE
    SUBROUTINE SUB2 (A,B)
      OPTIONAL :: B
    END SUBROUTINE
  END INTERFACE
  OPTIONAL :: Y
  IF (PRESENT(Y)) THEN          ! Reference to Y conditional
    X = X + Y                   ! on its presence
  ENDIF
  CALL SUB2(X,Y)
END SUBROUTINE

SUBROUTINE SUB2 (A,B)
  OPTIONAL :: B                 ! B and Y are argument associated,
  IF (PRESENT(B)) THEN          ! even if Y is not present, in
    B = B * A                   ! which case, B is also not present
    PRINT*, B
  ELSE
    A = A**2
    PRINT*, A
  ENDIF
END SUBROUTINE
```

Related Information

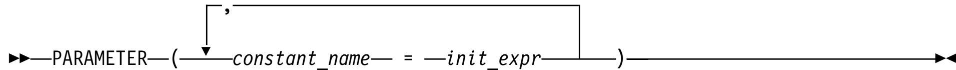
- "Optional Dummy Arguments" on page 167
- "Interface Concepts" on page 142
- "PRESENT(A)" on page 604
- "Dummy Arguments" on page 162

PARAMETER

Purpose

The **PARAMETER** attribute specifies names for constants.

Format



init_expr
: is an initialization expression

Rules

A named constant must have its type, shape, and parameters specified in a previous specification statement in the same scoping unit or be declared implicitly. If a named constant is implicitly typed, its appearance in any subsequent type declaration statement or attribute specification statement must confirm the implied type and any parameter values.

You can define *constant_name* only once with a **PARAMETER** attribute in a scoping unit.

A named constant that is specified in the initialization expression must have been previously defined (possibly in the same **PARAMETER** or type declaration statement, if not in a previous statement) or made accessible through use or host association.

The initialization expression is assigned to the named constant using the rules for intrinsic assignment. If the named constant is of type character and it has inherited length, it takes on the length of the initialization expression.

Attributes Compatible with the PARAMETER Attribute

- DIMENSION
- PRIVATE
- PUBLIC

Examples

```
REAL, PARAMETER :: TWO=2.0

COMPLEX      XCONST
REAL         RPART,IPART
PARAMETER    (RPART=1.1,IPART=2.2)
PARAMETER    (XCONST = (RPART,IPART+3.3))

CHARACTER*2, PARAMETER :: BB='  '

   ⋮

END
```

Related Information

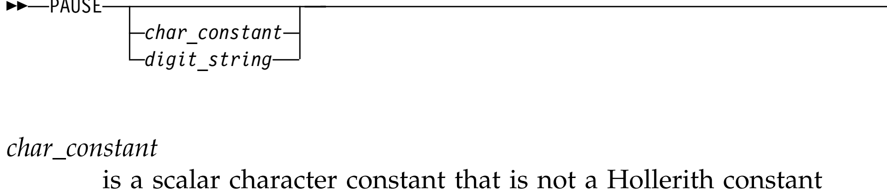
- “Initialization Expressions” on page 87
- “Data Objects” on page 28

PAUSE

Purpose

The **PAUSE** statement temporarily suspends the execution of a program and prints the keyword **PAUSE** and, if specified, a character constant or digit string to unit 0.

Format



char_constant
: is a scalar character constant that is not a Hollerith constant

digit_string
: is a string of one to five digits

Rules

After execution of a **PAUSE** statement, processing continues when you press the **Enter** key. If unit 5 is not connected to the terminal, the **PAUSE** statement does not suspend execution.

The **PAUSE** statement has been deleted in Fortran 95.

Examples

```
PAUSE 'Ensure backup tape is in tape drive'
PAUSE 10             ! Output:  PAUSE 10
```

Related Information

- “Deleted Features” on page 735

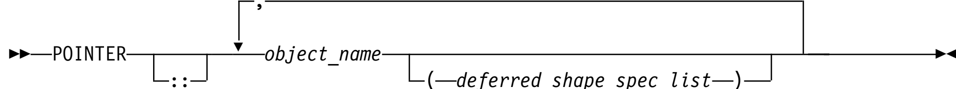
POINTER (Fortran 90)

Purpose

The **POINTER** attribute designates objects as pointer variables.

The term *pointer* refers to objects with the Fortran 90 **POINTER** attribute. The integer **POINTER** statement provides details on what was documented in previous versions of XL Fortran as the **POINTER** statement; these pointers are now referred to as *integer pointers*.

Format



deferred_shape_spec
is a colon (:), where each colon represents a dimension

Rules

object_name refers to a data object or function result. If *object_name* is declared elsewhere in the scoping unit with the **DIMENSION** attribute, the array specification must be a *deferred_shape_spec_list*.

object_name must not appear in an integer **POINTER**, **NAMELIST**, or **EQUIVALENCE** statement. If *object_name* is a component of a derived-type definition, any variables declared with that type cannot be specified in an **EQUIVALENCE**, **DATA**, or **NAMELIST** statement.

Pointer variables can appear in common blocks and block data program units.

To ensure that Fortran 90 pointers are thread-specific, do not specify either the **SAVE** or **STATIC** attribute for the pointer. These attributes are either specified explicitly by the user, or implicitly through the use of the **-qsave** compiler option. Note, however, that if a non-static pointer is used in a pointer assignment statement where the target is static, all references to the pointer are, in fact, references to the static, shared target.

An object having a component with the **POINTER** attribute can itself have the **TARGET**, **INTENT**, or **ALLOCATABLE** attributes, although it cannot appear in a data transfer statement.

Attributes Compatible with the POINTER Attribute

- AUTOMATIC
- DIMENSION
- OPTIONAL
- PRIVATE
- PUBLIC
- SAVE
- STATIC
- VOLATILE

These attributes apply only to the pointer itself, not to any associated targets, except for the **DIMENSION** attribute, which applies to associated targets.

Examples

Example1:

```
INTEGER, POINTER :: PTR(:)
INTEGER, TARGET :: TARG(5)
PTR => TARG                   ! PTR is associated with TARG and is
                              !   assigned an array specification of (5)

PTR(1) = 5                    ! TARG(1) has value of 5
PRINT *, FUNC()
CONTAINS
  REAL FUNCTION FUNC()
    POINTER :: FUNC           ! Function result is a pointer

   ⋮

  END FUNCTION
END
```

Example 2: Fortran 90 pointers and threadsafing

```
FUNCTION MYFUNC(ARG)                 ! MYPTR is thread-specific.
INTEGER, POINTER :: MYPTR            !  every thread that invokes
                                     !  'MYFUNC' will allocate a
ALLOCATE(MYPTR)                      !  new piece of storage that
MYPTR = ARG                          !  is only accessible within
   ⋮
                                     !  that thread.
ANYVAR = MYPTR
END FUNCTION
```

Related Information

- “Pointer Assignment” on page 117
- “TARGET” on page 398
- “ALLOCATED(ARRAY)” on page 529
- “DEALLOCATE” on page 280
- “Pointer Association” on page 139
- “Deferred-Shape Arrays” on page 69

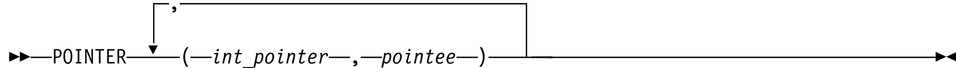
POINTER (integer)

Purpose

The integer **POINTER** statement specifies that the value of the variable *int_pointer* is to be used as the address for any reference to *pointee*.

The name of this statement has been changed from **POINTER** to integer **POINTER** to distinguish it from the Fortran 90 **POINTER** statement. The functionality and syntax for this statement remain the same as in previous releases of XL Fortran; only the name has changed.

Format



int_pointer
is the name of an integer pointer variable

pointee is a variable name or array declarator

Rules

The compiler does not allocate storage for the pointee. Storage is associated with the pointee at execution time by the assignment of the address of a block of storage to the pointer. The pointee can become associated with either static or dynamic storage. A reference to a pointee requires that the associated pointer be defined.

An integer pointer is a scalar variable of type **INTEGER(4)** in 32-bit mode and type **INTEGER(8)** in 64-bit mode that cannot have a type explicitly assigned to it. You can use integer pointers in any expression or statement in which a variable of the same type as the integer pointer can be used. You can assign any data type to a pointee, but you cannot assign a storage class or initial value to a pointee.

An actual array that appears as a pointee in an integer **POINTER** statement is called a pointee array. You can dimension a pointee array in a type declaration statement, a **DIMENSION** statement, or in the integer **POINTER** statement itself.

If you specify the **-qddim** compiler option, a pointee array that appears in a main program can also have an adjustable array specification. In main programs and subprograms, the dimension size is evaluated when the pointee is referenced (dynamic dimensioning).

If you do not specify the **-qddim** compiler option, a pointee array that appears in a subprogram can have an adjustable array specification, and the dimension size is evaluated on entrance to the subprogram, not when the pointee is evaluated.

The following constraints apply to the definition and use of pointees and integer pointers:
- A pointee cannot be zero-sized.
- A pointee can be scalar, an assumed-sized array or an explicit-shape array.
- A pointee cannot appear in a **COMMON**, **DATA**, **NAMELIST**, or **EQUIVALENCE** statement.

- A pointee cannot have the following attributes: **EXTERNAL**, **ALLOCATABLE**, **POINTER**, **TARGET**, **INTRINSIC**, **INTENT**, **OPTIONAL**, **SAVE**, **STATIC**, **AUTOMATIC**, or **PARAMETER**.
- A pointee cannot be a dummy argument and therefore cannot appear in a **FUNCTION**, **SUBROUTINE**, or **ENTRY** statement.
- A pointee cannot be an automatic object.
- A pointee cannot be a generic interface block name.
- A pointee that is of derived type must be of sequence derived type.
- A function value cannot be a pointee.
- An integer pointer cannot be pointed to by another pointer. (A pointer cannot be a pointee.)
- An integer pointer cannot have the following attributes: **DIMENSION**, **POINTER**, **TARGET**, **PARAMETER**, **ALLOCATABLE**, **EXTERNAL**, and **INTRINSIC**.
- An integer pointer cannot appear as a **NAMELIST** group name.
- An integer pointer cannot be a procedure.

Examples

```
INTEGER A,B
POINTER (P,I)
IF (A<>0) THEN
  P=LOC(A)
ELSE
  P=LOC(B)
ENDIF
I=0         ! Assigns 0 to either A or B, depending on A's value
END
```

Related Information

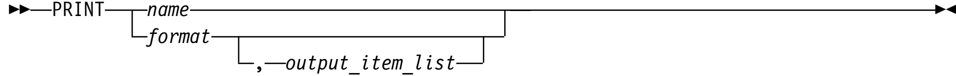
- “Integer Pointer Association” on page 140
- “LOC(X)” on page 581
- ″**-qddim** Option″ in the *User's Guide*

PRINT

Purpose

The **PRINT** statement is a data transfer output statement.

Format



name is a namelist group name

output_item
is an output list item. An output list specifies the data to be transferred. An output list item can be:

- A variable. An array is treated as if all of its elements were specified in the order they are arranged in storage.

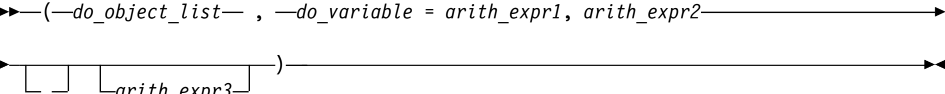
  A pointer must be associated with a target, and an allocatable array must be allocated. A derived-type object cannot have any ultimate component that is outside the scoping unit of this statement. The evaluation of *output_item* cannot result in a derived-type object that contains a pointer. The structure components of a structure in a formatted statement are treated as if they appear in the order of the derived-type definition; in an unformatted statement, the structure components are treated as a single value in their internal representation (including padding).
- An expression.
- An implied-**DO** list, as described on page 370

format is a format specifier that specifies the format to be used in the output operation. *format* is a format identifier that can be:

- The statement label of a **FORMAT** statement. The **FORMAT** statement must be in the same scoping unit.
- The name of a scalar **INTEGER(4)** or **INTEGER(8)** variable that was assigned the statement label of a **FORMAT** statement. The **FORMAT** statement must be in the same scoping unit.

  Fortran 95 does not permit assigning of a statement label.
- A character constant. It cannot be a Hollerith constant. It must begin with a left parenthesis and end with a right parenthesis. Only the format codes described in the **FORMAT** statement can be used between the parentheses. Blank characters can precede the left parenthesis, or follow the right parenthesis.
- A character variable that contains character data whose leftmost character positions constitute a valid format. A valid format begins with a left parenthesis and ends with a right parenthesis. Only the format codes listed under “FORMAT” on page 315 can be used between the parentheses. Blank characters can precede the left parenthesis, or follow the right parenthesis.
- An array of noncharacter intrinsic type.
- Any character expression, except one involving concatenation of an operand that specifies inherited length, unless the operand is the name of a constant.
- An asterisk, specifying list-directed formatting.
- A namelist specifier that specifies the name of a namelist list that you have previously defined.

Implied-DO List



do_object
: is an output list item

do_variable
: is a named scalar variable of type integer or real

arith_expr1, *arith_expr2*, and *arith_expr3*
: are scalar numeric expressions

The range of an implied-**DO** list is the list *do_object_list*. The iteration count and the values of the **DO** variable are established from *arith_expr1*, *arith_expr2*, and *arith_expr3*, the same as for a **DO** statement. When the implied-**DO** list is executed, the items in the *do_object_list* are specified once for each iteration of the implied-**DO** list, with the appropriate substitution of values for any occurrence of the **DO** variable.

Examples

```
   PRINT 10, A,B,C
10 FORMAT (E4.2,G3.2E1,B3)
```

Related Information

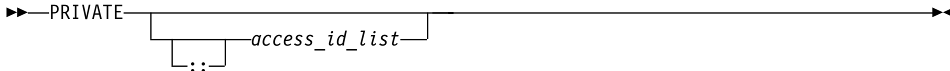
- "Chapter 8. Input/Output Concepts" on page 183
- "Chapter 9. Input/Output Formatting" on page 201
- "Deleted Features" on page 735

PRIVATE

Purpose

The **PRIVATE** attribute specifies that a module entity is not accessible outside the module through use association.

Format



access_id
: is a generic specification or the name of a variable, procedure, derived type, constant, or namelist group

Rules

The **PRIVATE** attribute can appear only in the scope of a module.

Although multiple **PRIVATE** statements may appear in a module, only one statement that omits an *access_id_list* is permitted. A **PRIVATE** statement without an *access_id_list* sets the default accessibility to private for all potentially accessible entities in the module. If the module contains such a statement, it cannot also include a **PUBLIC** statement without an *access_id_list*. If the module does not contain such a statement, the default accessibility is public. Entities whose accessibility is not explicitly specified have default accessibility.

A procedure that has a generic identifier that is public is accessible through that identifier, even if its specific identifier is private. If a module procedure contains a private dummy argument or function result whose type has private accessibility, the module procedure must be declared to have private accessibility and must not have a generic identifier that has public accessibility.

If a **PRIVATE** statement is specified within a derived-type definition, all the components of the derived type become private.

A structure must be private if its derived type is private. A namelist group must be private if it contains any object that is private or contains private components. A derived type that has a component of derived type that is private must itself be private or have private components. A subprogram must be private if any of its arguments are of a derived type that is private. A function must be private if its result variable is of a derived type that is private.

Attributes Compatible with the PRIVATE Attribute

- ALLOCATABLE
- DIMENSION
- EXTERNAL
- INTRINSIC
- PARAMETER
- POINTER
- SAVE
- STATIC
- TARGET
- VOLATILE

Examples

```
MODULE MC
   PUBLIC                    ! Default accessibility declared as public
   INTERFACE GEN
      MODULE PROCEDURE SUB1, SUB2
   END INTERFACE
   PRIVATE SUB1              ! SUB1 declared as private
   CONTAINS
      SUBROUTINE SUB1(I)
```

```
          INTEGER I
          I = I + 1
       END SUBROUTINE SUB1
       SUBROUTINE SUB2(I,J)
          I = I + J
       END SUBROUTINE
END MODULE MC

PROGRAM ABC
   USE MC
   K = 5
   CALL GEN(K)                ! SUB1 referenced because GEN has public
                              ! accessibility and appropriate argument
                              ! is passed
   CALL SUB2(K,4)
   PRINT *, K                 ! Value printed is 10
END PROGRAM
```

Related Information

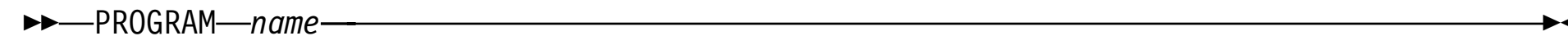
- "Derived Types" on page 39
- "Modules" on page 153
- "PUBLIC" on page 373

PROGRAM

Purpose

The **PROGRAM** statement specifies that a program unit is a main program, the program unit that receives control from the system when the executable program is invoked at run time.

Format



name is the name of the main program in which this statement appears

Rules

The **PROGRAM** statement is optional.

If specified, the **PROGRAM** statement must be the first statement of the main program.

If a program name is specified in the corresponding **END** statement, it must match *name*.

The program name is global to the executable program. This name must not be the same as the name of any common block, external procedure, or any other program unit in that executable program, or as any name that is local to the main program.

The name has no type, and it must not appear in any type declaration or specification statements. You cannot refer to a main program from a subprogram or from itself.

Examples

```
PROGRAM DISPLAY_NUMBER_2
   INTEGER A
   A = 2
   PRINT *, A
END PROGRAM DISPLAY_NUMBER_2
```

Related Information

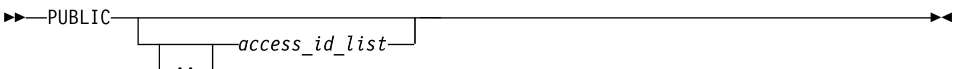
“Main Program” on page 152

PUBLIC

Purpose

The **PUBLIC** attribute specifies that a module entity can be accessed by other program units through use association.

Format



access_id
: is a generic specification or the name of a variable, procedure, derived type, constant, or namelist group

Rules

The **PUBLIC** attribute can appear only in the scope of a module.

Although multiple **PUBLIC** statements can appear in a module, only one statement that omits an *access_id_list* is permitted. A **PUBLIC** statement without an *access_id_list* sets the default accessibility to public for all potentially accessible entities in the module. If the module contains such a statement, it cannot also include a **PRIVATE** statement without an *access_id_list*. If the module does not contain such a statement, the default accessibility is public. Entities whose accessibility is not explicitly specified have default accessibility.

A procedure that has a generic identifier that is public is accessible through that identifier, even if its specific identifier is private. If a module procedure contains a private dummy argument or function result whose type has private accessibility, the module procedure must be declared to have private accessibility and must not have a generic identifier that has public accessibility.

Although an entity with public accessibility cannot have the **STATIC** attribute, public entities in a module are unaffected by **IMPLICIT STATIC** statements in the module.

Attributes Compatible with the PUBLIC Attribute

- ALLOCATABLE
- DIMENSION
- EXTERNAL
- INTRINSIC
- PARAMETER
- POINTER
- SAVE
- TARGET
- VOLATILE

Examples

```
MODULE MC
   PRIVATE                        ! Default accessibility declared as private
   PUBLIC GEN                     ! GEN declared as public
   INTERFACE GEN
      MODULE PROCEDURE SUB1
   END INTERFACE
   CONTAINS
      SUBROUTINE SUB1(I)
         INTEGER I
         I = I + 1
      END SUBROUTINE SUB1
END MODULE MC
PROGRAM ABC
   USE MC
   K = 5
   CALL GEN(K)                    ! SUB1 referenced because GEN has public
                                  !   accessibility and appropriate argument
                                  !   is passed
   PRINT *, K                     ! Value printed is 6
END PROGRAM
```

Related Information

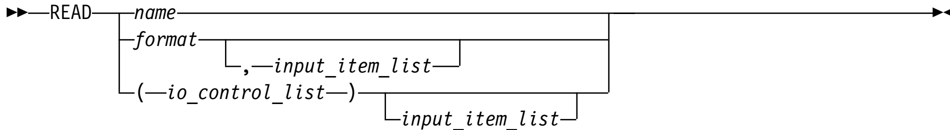
- "PRIVATE" on page 370
- "Modules" on page 153

READ

Purpose

The **READ** statement is the data transfer input statement.

Format



format is a format identifier, described below under **FMT=***format*. In addition, it cannot be a Hollerith constant.

name is a namelist group name

input_item
is an input list item. An input list specifies the data to be transferred. An input list item can be:

- A variable name, but not for an assumed-size array. An array is treated as if all of its elements were specified in the order they are arranged in storage.

  A pointer must be associated with a definable target, and an allocatable array must be allocated. A derived-type object cannot have any ultimate component that is outside the scoping unit of this statement. The evaluation of *input_item* cannot result in a derived-type object that contains a pointer. The structure components of a structure in a formatted statement are treated as if they appear in the order of the derived-type definition; in an unformatted statement, the structure components are treated as a single value in their internal representation (including padding).
- An implied-**DO** list, as described under "Implied-DO List" on page 379.

io_control
is a list that must contain one unit specifier (**UNIT=**) and can also contain one of each of the other valid specifiers described below.

[UNIT=] *u*
is a unit specifier that specifies the unit to be used in the input operation. *u* is an external unit identifier or internal file identifier.

An external unit identifier refers to an external file. It is one of the following:
- An integer expression whose value is in the range 0 through 2,147,483,647.
- An asterisk, which identifies external unit 5 and is preconnected to standard input.

An internal file identifier refers to an internal file. It is the name of a character variable that cannot be an array section with a vector subscript.

If the optional characters **UNIT=** are omitted, *u* must be the first item in *io_control_list*. If the optional characters **UNIT=** are specified, either the optional characters **FMT=** or the optional characters **NML=** must also be present.

[FMT=] *format*
is a format specifier that specifies the format to be used in the input operation. *format* is a format identifier that can be:

- The statement label of a **FORMAT** statement. The **FORMAT** statement must be in the same scoping unit.
- The name of a scalar **INTEGER(4)** or **INTEGER(8)** variable that was assigned the statement label of a **FORMAT** statement. The **FORMAT** statement must be in the same scoping unit.

  Fortran 95 does not permit assigning of a statement label.
- A character constant. It must begin with a left parenthesis and end with a right parenthesis. Only the format codes described in the **FORMAT** statement can be used between the parentheses. Blank characters can precede the left parenthesis, or follow the right parenthesis.
- A character variable that contains character data whose leftmost character positions constitute a valid format. A valid format begins with a left parenthesis and ends with a right parenthesis. Only the format codes listed under "FORMAT" on page 315 can be used between the parentheses. Blank characters can precede the left parenthesis or follow the right parenthesis. If *format* is an array element, the format identifier must not exceed the length of the array element.
- An array of noncharacter intrinsic type. The data must be a valid format identifier as described under character array.
- Any character expression, except one involving concatenation of an operand that specifies inherited length, unless the operand is the name of a constant.
- An asterisk, specifying list-directed formatting.
- A namelist specifier that specifies the name of a namelist list that you have previously defined.

If the optional characters **FMT=** are omitted, *format* must be the second item in *io_control_list* and the first item must be the unit specifier with the optional characters **UNIT=** omitted. Both **NML=** and **FMT=** cannot be specified in the same input statement.

REC= *integer_expr*
is a record specifier that specifies the number of the record to be read in a file connected for direct access. The **REC=** specifier is only permitted for direct input. *integer_expr* is an integer expression whose value is positive. A record specifier is not valid if list-directed or namelist formatting is used and if the unit specifier specifies an internal file. The **END=** specifier can appear concurrently. The record specifier represents the relative position of a record within a file. The relative position number of the first record is 1.

ID= *integer_variable*
indicates that the data transfer is to be done asynchronously. The *integer_variable* is a scalar of type **INTEGER(4)** or default integer. If no error is encountered, the *integer_variable* is defined with a value after executing the asynchronous data transfer statement. This value must be used in the matching **WAIT** statement.

Asynchronous data transfer must either be direct unformatted or sequential unformatted. Asynchronous I/O to internal files is prohibited. Asynchronous I/O to raw character devices (for example, to tapes or raw logical volumes) is prohibited. The *integer_variable* must not be associated with any entity in the data transfer I/O list, or with a *do_variable* of an *io_implied_do* in the data transfer I/O list. If the *integer_variable* is an array element reference, its subscript values must not be affected by the data transfer, the *io_implied_do* processing, or the definition or evaluation of any other specifier in the *io_control_spec*.

IOSTAT= *ios*
is an input/output status specifier that specifies the status of the input/output operation. *ios* is a variable of type **INTEGER(4)** or default integer. Coding the **IOSTAT=** specifier suppresses error messages. When the statement finishes execution, *ios* is defined with:
- A zero value if no error condition, end-of-file condition, or end-of-record condition occurs.
- A positive value if an error occurs.
- A negative value if an end-of-file condition is encountered and no error occurs.
- A negative value that is different from the end-of-file value if an end-of-record condition occurs and no error condition or end-of-file condition occurs.

ERR= *stmt_label*
is an error specifier that specifies the statement label of an executable statement to which control is to transfer in the case of an error. Coding the **ERR=** specifier suppresses error messages.

END= *stmt_label*
is an end-of-file specifier that specifies a statement label at which the program is to continue if an endfile record is encountered and no error occurs. An external file is positioned after the endfile record; the **IOSTAT=** specifier, if present, is assigned a negative value; and the **NUM=** specifier, if present, is assigned an integer value. If an error occurs and the statement contains the **SIZE=** specifier, the specified variable becomes defined with an integer value. Coding the **END=** specifier suppresses the error message for end-of-file. This specifier can be specified for a unit connected for either sequential or direct access.

NUM= *integer_variable*
is a number specifier that specifies the number of bytes of data transmitted between the I/O list and the file. *integer_variable* is a scalar variable name of type **INTEGER(4)**, type **INTEGER(8)** in 64-bit, or type default integer. The **NUM=** specifier is only permitted for unformatted output. Coding the **NUM** parameter suppresses the indication of an error that would occur if the number of bytes represented by the output list is greater than the number of bytes that can be written into the record. In this case, *integer_variable* is set to a value that is the maximum length record that can be written. Data from remaining output list items is not written into subsequent records.

[NML=] *name*
is a namelist specifier that specifies the name of a namelist list that you have previously defined. If the optional characters **NML=**are not specified, the namelist name must appear as the second parameter in the list and the first item must be the unit specifier with **UNIT=** omitted. If both **NML=**and **UNIT=**are specified, all the parameters can appear in any order. The **NML=** specifier is an alternative to **FMT=**; both **NML=** and **FMT=** cannot be specified in the same input statement.

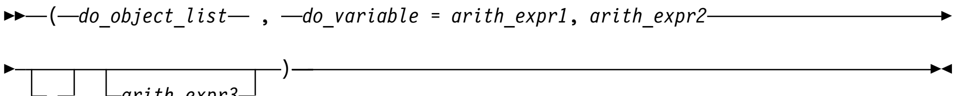
ADVANCE= *char_expr*
is an advance specifier that determines whether nonadvancing input occurs for this statement. *char_expr* is a scalar character expression that must evaluate to **YES** or **NO**. If **NO** is specified, nonadvancing input occurs. If **YES** is specified, advancing, formatted sequential input occurs. The default value is **YES**. **ADVANCE=** can be specified only in a formatted sequential **READ** statement with an explicit format specification that does not specify an internal file unit specifier.

SIZE= *count*
is a character count specifier that determines how many characters are transferred by data edit descriptors during execution of the current input statement. *count* is a scalar variable of type default integer, type

INTEGER(4), or type **INTEGER(8)** in 64-bit. Blanks that are inserted as padding are not included in the count.

EOR= *stmt_label*
is an end-of-record specifier. If the specifier is present, an end-of-record condition occurs, and no error condition occurs during execution of the statement. If **PAD=** exists, the following also occur:
1. If the **PAD=** specifier has the value **YES**, the record is padded with blanks to satisfy the input list item and the corresponding data edit descriptor that requires more characters than the record contains.
2. Execution of the **READ** statement terminates.
3. The file specified in the **READ** statement is positioned after the current record.
4. If the **IOSTAT=** specifier is present, the specified variable becomes defined with a negative value different from an end-of-file value.
5. If the **SIZE=** specifier is present, the specified variable becomes defined with an integer value.
6. Execution continues with the statement containing the statement label specified by the **EOR=** specifier.
7. End-of-record messages are suppressed.

Implied-DO List



do_object
is an output list item

do_variable
is a named scalar variable of type integer or real

arith_expr1, *arith_expr2*, and *arith_expr3*
are scalar numeric expressions

The range of an implied-**DO** list is the list *do_object_list*. The iteration count and the values of the **DO** variable are established from *arith_expr1*, *arith_expr2*, and *arith_expr3*, the same as for a **DO** statement. When the implied-**DO** list is executed, the items in the *do_object_list* are specified once for each iteration of the implied-**DO** list, with the appropriate substitution of values for any occurrence of the **DO** variable.

The **DO** variable or an associated data item must not appear as an input list item in the *do_object_list*, but can be read in the same **READ** statement outside of the implied-**DO** list.

Rules

Any statement label specified by the **ERR=**, **EOR=** and **END=** specifiers must refer to a branch target statement that appears in the same scoping unit as the **READ** statement.

If either the **EOR=** specifier or the **SIZE=** specifier is present, the **ADVANCE=** specifier must also be present and must have the value **NO**.

If a **NUM=** specifier is present, neither a format specifier nor a namelist specifier can be present.

Variables specified for the **IOSTAT=**, **SIZE=** and **NUM=** specifiers must not be associated with any input list item, namelist list item, or the **DO** variable of an implied-**DO** list. If such a specifier variable is an array element, its subscript values must not be affected by the data transfer, any implied-**DO** processing, or the definition or evaluation of any other specifier.

A **READ** statement without *io_control_list* specified specifies the same unit as a **READ** statement with *io_control_list* specified in which the external unit identifier is an asterisk.

If the **ERR=** and **IOSTAT=** specifiers are set and an error is encountered during a synchronous data transfer, transfer is made to the statement specified by the **ERR=** specifier and a positive integer value is assigned to *ios*.

If the **ERR=** or **IOSTAT=** specifiers are set and an error is encountered during an asynchronous data transfer, execution of the matching **WAIT** statement is not required.

If the **END=** or **IOSTAT=** specifiers are set and an end-of-file condition is encountered during an asynchronous data transfer, execution of the matching **WAIT** statement is not required.

If a conversion error is encountered and the **CNVERR** run-time option is set to **NO**, **ERR=** is not branched to, although **IOSTAT=** may be set.

If **IOSTAT=** and **ERR=** are not specified,
- The program stops if a severe error is encountered.
- The program continues to the next statement if a recoverable error is encountered and the **ERR_RECOVERY** run-time option is set to **YES**. If the option is set to **NO**, the program stops.

- The program continues to the next statement when a conversion error is encountered if the **ERR_RECOVERY** run-time option is set to **YES**. If the **CNVERR** run-time option is set to **YES**, conversion errors are treated as recoverable errors; if **CNVERR=NO**, they are treated as conversion errors.

Examples

```
INTEGER A(100)
CHARACTER*4 B
READ *, A(LBOUND(A,1):UBOUND(A,1))
READ (7,FMT='(A3)',ADVANCE='NO',EOR=100) B

   ⋮

100 PRINT *, 'end of record reached'
END
```

Related Information

- “Executing Data Transfer Statements Asynchronously” on page 189
- ″AIX Implementation Details of XL Fortran Input/Output″ in the *User's Guide*
- “Conditions and IOSTAT Values” on page 193
- “WRITE” on page 416
- “WAIT” on page 412
- “Chapter 8. Input/Output Concepts” on page 183
- ″Setting Runtime Options for Input/Output″ in the *User's Guide*
- “Deleted Features” on page 735

REAL

Purpose

A **REAL** type declaration statement specifies the length and attributes of objects and functions of type real. Initial values can be assigned to objects.

Format

where:

attr_spec
ALLOCATABLE **AUTOMATIC** **DIMENSION** (*array_spec*) **EXTERNAL** **INTENT** (*intent_spec*) **INTRINSIC** **OPTIONAL** **PARAMETER** **POINTER** **PRIVATE** **PUBLIC** **SAVE** **STATIC** **TARGET** **VOLATILE**

kind_selector

```
>>──┬─(─┬─────────┬─int_initialization_expr─)─┬──────────────────><
    │   └─KIND─ =─┘                            │
    └─ * ─int_literal_constant─────────────────┘
```

specifies the length of real entities: 4, 8 or 16. *int_literal_constant* cannot specify a kind type parameter.

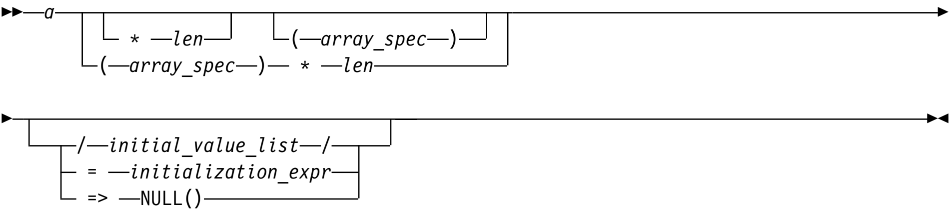
attr_spec
: For detailed information on rules about a particular attribute, refer to the statement of the same name.

intent_spec
: is either **IN**, **OUT**, or **INOUT**

::
: is the double colon separator. It is required if attributes are specified, = *initialization_expr* or => **NULL()** is used.

array_spec
: is a list of dimension bounds

entity_decl



a
: is an object name or function name. *array_spec* cannot be specified for a function name.

len
: overrides the length as specified in *kind_selector*, and cannot specify a kind type parameter. The entity length must be an integer literal constant that represents one of the permissible length specifications.

initial_value
: provides an initial value for the entity specified by the immediately preceding name.

initialization_expr
: provides an initial value, by means of an initialization expression, for the entity specified by the immediately preceding name

=> **NULL()**
: provides the initial value for the pointer object

Rules

Within the context of a derived type definition:

- If => appears in a component initialization, the **POINTER** attribute must appear in the *attr_spec_list*.
- If = appears in a component initialization, the **POINTER** attribute cannot appear in the component *attr_spec_list*.
- The compiler will evaluate *initialization_expr* within the scoping unit of the type definition.

If => appears for a variable, the object must have the **POINTER** attribute.

If *initialization_expr* appears for a variable, the object cannot have the **POINTER** attribute.

Entities in type declaration statements are constrained by the rules of any attributes specified for the entities, as detailed in the corresponding attribute statements.

The type declaration statement overrides the implicit type rules in effect. You can use a type declaration statement that confirms the type of an intrinsic function. The appearance of a generic or specific intrinsic function name in a type declaration statement does not cause the name to lose its intrinsic property.

An object cannot be initialized in a type declaration statement if it is a dummy argument, an allocatable array, a pointer, a function result, an object in blank common, an integer pointer, an external name, an intrinsic name, or an automatic object. Nor can an object be initialized if it has the **AUTOMATIC** attribute. The object may be initialized if it appears in a named common block in a block data program unit or if it appears in a named common block in a module.

In Fortran 95, a pointer can be initialized. Pointers can only be initialized by the use of **=> NULL()**.

The specification expression of an *array_spec* can be a nonconstant expression if the specification expression appears in an interface body or in the specification part of a subprogram. Any object being declared that uses this nonconstant expression and is not a dummy argument or a pointee is called an *automatic object*.

An attribute cannot be repeated in a given type declaration statement, nor can an entity be explicitly given the same attribute more than once in a scoping unit.

initialization_expr must be specified if the statement contains the **PARAMETER** attribute. If the entity you are declaring is a variable, and *initialization_expr* or **NULL()** is specified, the variable is initially defined. If the entity you are declaring is a derived type component, and *initialization_expr* or **NULL()** is specified, the derived type has default initialization. *a* becomes defined with the value determined by *initialization_expr*, in accordance with the rules for intrinsic assignment. If the entity is an array, its shape must be specified either in the type declaration statement or in a previous specification statement in the same scoping unit. A variable or variable subobject cannot be initialized more than once. If *a* is a variable, the presence of *initialization_expr* or => **NULL()** implies that *a* is a saved object, except for an object in a named common block. The initialization of an object could affect the fundamental storage class of an object.

An *array_spec* specified in the *entity_decl* takes precedence over the *array_spec* in the **DIMENSION** attribute.

An array function result that does not have the **POINTER** attribute must have an explicit-shape array specification.

If the entity declared is a function, it must not have an accessible explicit interface unless it is an intrinsic function.

If `T` or `F`, defined previously as the name of a constant, appears in a type declaration statement, it is no longer an abbreviated logical constant but the name of the named constant.

Examples

```
REAL(8), POINTER :: RPTR
REAL(8), TARGET  :: RTAR
```

Related Information

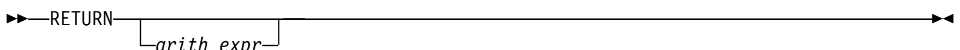
- "Real" on page 30
- "Initialization Expressions" on page 87
- "How Type Is Determined" on page 52, for details on the implicit typing rules
- "Array Declarators" on page 65
- "Automatic Objects" on page 29
- "Storage Classes for Variables" on page 59
- "DATA" on page 277, for details on initial values

RETURN

Purpose

The **RETURN** statement:
- In a function subprogram, ends the execution of the subprogram and returns control to the referencing statement. The value of the function is available to the referencing procedure.
- In a subroutine subprogram, ends the subprogram and transfers control to the first executable statement after the procedure reference or to an alternate return point, if one is specified.
- In the main program, ends execution of the executable program.

Format



arith_expr
is a scalar integer, real, or complex expression. If the value of the

expression is noninteger, it is converted to **INTEGER(4)** before use. *arith_expr* cannot be a Hollerith constant.

Rules

arith_expr can be specified in a subroutine subprogram only, and it specifies an alternate return point. Letting *m* be the value of *arith_expr*, if 1 ≤ *m* ≤ the number of asterisks in the **SUBROUTINE** or **ENTRY** statement, the *m*th asterisk in the dummy argument list is selected. Control then returns to the invoking procedure at the statement whose statement label is specified as the *m*th alternate return specifier in the **CALL** statement. For example, if the value of *m* is 5, control returns to the statement whose statement label is specified as the fifth alternate return specifier in the **CALL** statement.

If *arith_expr* is omitted or if its value (*m*) is not in the range 1 through the number of asterisks in the **SUBROUTINE** or **ENTRY** statement, a normal return is executed. Control returns to the invoking procedure at the statement following the **CALL** statement.

Executing a **RETURN** statement terminates the association between the dummy arguments of the subprogram and the actual arguments supplied to that instance of the subprogram. All entities local to the subprogram become undefined, except as noted under "Events Causing Undefinition" on page 55.

A subprogram can contain more than one **RETURN** statement, but it does not require one. An **END** statement in a function or subroutine subprogram has the same effect as a **RETURN** statement.

Examples

```
CALL SUB(A,B)
CONTAINS
  SUBROUTINE SUB(A,B)
    INTEGER :: A,B
    IF (A.LT.B)
      RETURN           ! Control returns to the calling procedure
    ELSE
   .
   .
   .
    END IF
  END SUBROUTINE
END
```

Related Information

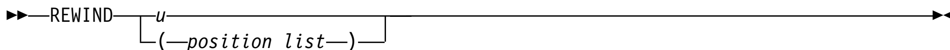
- "Asterisks as Dummy Arguments" on page 172
- "Actual Argument Specification" on page 161 for a description of alternate return points
- "Events Causing Undefinition" on page 55

REWIND

Purpose

The **REWIND** statement positions an external file connected for sequential access at the beginning of the first record of the file.

Format



u
is an external unit identifier. The value of *u* must not be an asterisk or a Hollerith constant.

position_list
is a list that must contain one unit specifier ([**UNIT=**]*u*) and can also contain one of each of the other valid specifiers. The valid specifiers are:

[UNIT=] *u*
is a unit specifier in which *u* must be an external unit identifier whose value is not an asterisk. An external unit identifier refers to an external file that is represented by a scalar integer expression, whose value is in the range 1 through 2,147,483,647. If the optional characters **UNIT=** are omitted, *u* must be the first item in *position_list*.

IOSTAT= *ios*
is an input/output status specifier that specifies the status of the input/output operation. *ios* is a scalar variable of type **INTEGER(4)** or default integer. When the **REWIND** statement finishes executing, *ios* is defined with:
- A zero value if no error condition occurs
- A positive value if an error occurs.

ERR= *stmt_label*
is an error specifier that specifies the statement label of an executable statement in the same scoping unit to which control is to transfer in the case of an error. Coding the **ERR=** specifier suppresses error messages.

Rules

If the unit is not connected, an implicit **OPEN** specifying sequential access is performed to a default file named **fort.***n*, where *n* is the value of *u* with leading zeros removed. If the external file connected to the specified unit does not exist, the **REWIND** statement has no effect. If it exists, an end-of-file marker is created, if necessary, and the file is positioned at the beginning of the first record. If the file is already positioned at its initial point, the **REWIND** statement has no effect. The **REWIND** statement causes a

subsequent **READ** or **WRITE** statement referring to *u* to read data from or write data to the first record of the external file associated with *u*.

If the **ERR=** and **IOSTAT=** specifiers are set and an error is encountered, transfer is made to the statement specified by the **ERR=** specifier and a positive integer value is assigned to *ios*.

If **IOSTAT=** and **ERR=** are not specified,
- the program stops if a severe error is encountered.
- the program continues to the next statement if a recoverable error is encountered and the **ERR_RECOVERY** run-time option is set to **YES**. If the option is set to **NO**, the program stops.

Examples

```
REWIND (9, IOSTAT=IOSS)
```

Related Information

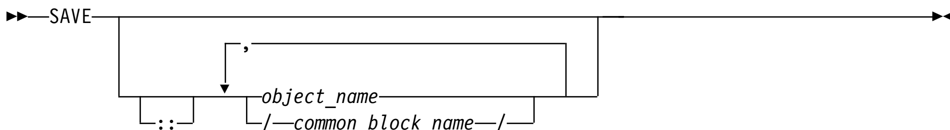
- "Conditions and IOSTAT Values" on page 193
- "Chapter 8. Input/Output Concepts" on page 183
- "Setting Runtime Options for Input/Output" in the *User's Guide*

SAVE

Purpose

The **SAVE** attribute specifies the names of objects and named common blocks whose definition status you want to retain after control returns from the subprogram where you define the variables and named common blocks.

Format



Rules

A **SAVE** statement without a list is treated as though it contains the names of all common items and local variables in the scoping unit. A common block name having the **SAVE** attribute has the effect of specifying all the entities in that named common block.

Within a function or subroutine subprogram, a variable whose name you specify with the **SAVE** attribute does not become undefined as a result of a **RETURN** or **END** statement in the subprogram.

object_name cannot be the name of a dummy argument, pointee, procedure, automatic object, or common block entity.

If a local entity specified with the **SAVE** attribute (and not in a common block) is in a defined state at the time that a **RETURN** or **END** statement is encountered in a subprogram, that entity is defined with the same value at the next reference of that subprogram. Saved objects are shared by all instances of the subprogram.

XL Fortran permits function results to have the **SAVE** attribute. To indicate that a function result is to have the **SAVE** attribute, the function result name must be explicitly specified with the **SAVE** attribute. That is, a **SAVE** statement without a list does not provide the **SAVE** attribute for the function result.

Variables declared as **SAVE** are shared amongst threads. To thread-safe an application that contains shared variables, you must either serialize access to the static data using locks, or make the data thread-specific. One method of making the data thread-specific is to move the static data into a named **COMMON** block that has been declared **THREADLOCAL**. The **Pthreads** library module provides mutexes to allow you to serialize access to the data using locks. See "Chapter 15. Pthreads Library Module" on page 677 for more information. The *lock_name* attribute on the **CRITICAL** directive also provides the ability to serialize access to data. See "CRITICAL / END CRITICAL" on page 453 for more information. The **THREADLOCAL** directive ensures that common blocks are local to each thread. See "THREADLOCAL" on page 507 for more information.

Attributes Compatible with the SAVE Attribute

- ALLOCATABLE
- DIMENSION
- POINTER
- PRIVATE
- PUBLIC
- STATIC
- TARGET
- VOLATILE

Examples

```
LOGICAL :: CALLED=.FALSE.
CALL SUB(CALLED)
CALLED=.TRUE.
CALL SUB(CALLED)
CONTAINS
  SUBROUTINE SUB(CALLED)
    INTEGER, SAVE :: J
    LOGICAL :: CALLED
    IF (CALLED.EQV..FALSE.) THEN
      J=2
    ELSE
      J=J+1
```

```
      ENDIF
      PRINT *, J                    ! Output on first call is 2
                                    ! Output on second call is 3
    END SUBROUTINE
  END
```

Related Information

- "COMMON" on page 267
- "THREADLOCAL" on page 507
- "Definition Status of Variables" on page 53
- "Storage Classes for Variables" on page 59
- Item 2 under "Appendix A. Compatibility Across Standards" on page 731

SELECT CASE

Purpose

The **SELECT CASE** statement is the first statement of a **CASE** construct. It provides a concise syntax for selecting, at most, one of a number of statement blocks for execution.

Format

```
>>-+-----------------------+--SELECT CASE--(--case_expr--)-----------><
   '-case_construct_name--:-'
```

case_construct_name
: is a name that identifies the **CASE** construct

case_expr
: is a scalar expression of type integer, character or logical

Rules

When a **SELECT CASE** statement is executed, the *case_expr* is evaluated. The resulting value is called the case index, which is used for evaluating control flow within the case construct.

If the *case_construct_name* is specified, it must appear on the **END CASE** statement and optionally on any **CASE** statements within the construct.

The *case_expr* must not be a typeless constant or a **BYTE** data object.

Examples

```
ZERO: SELECT CASE(N)         ! start of CASE construct ZERO

  CASE DEFAULT ZERO
       OTHER: SELECT CASE(N) ! start of CASE construct OTHER
          CASE(:-1)
             SIGNUM = -1
```

```
                CASE(1:) OTHER
                    SIGNUM = 1
             END SELECT OTHER
        CASE (0)
          SIGNUM = 0

      END SELECT ZERO
```

Related Information

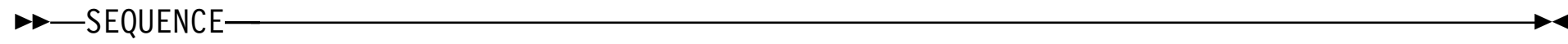
- "CASE Construct" on page 123
- "CASE" on page 258
- "END (Construct)" on page 298, for details on the **END SELECT** statement

SEQUENCE

Purpose

The **SEQUENCE** statement specifies that the order of the components in a derived-type definition establishes the storage sequence for objects of that type. Such a type becomes a *sequence derived type*.

Format



Rules

The **SEQUENCE** statement can be specified only once in a derived-type definition.

If a component of a sequence derived type is of derived type, that derived type must also be a sequence derived type.

The size of a sequence derived type is equal to the number of bytes of storage needed to hold all of the components of that derived type.

Use of sequence derived types can lead to misaligned data, which can adversely affect the performance of a program.

Examples

```
TYPE PERSON
  SEQUENCE
  CHARACTER*1 GENDER     ! Offset 0
  INTEGER(4) AGE         ! Offset 1
  CHARACTER(30) NAME     ! Offset 5
END TYPE PERSON
```

Related Information

- "Derived Types" on page 39
- "Derived Type" on page 282

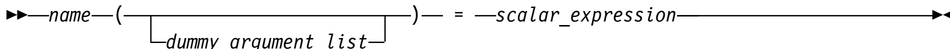
- "END TYPE" on page 301

Statement Function

Purpose

A statement function defines a function in a single statement.

Format



name is the name of the statement function. It must not be supplied as a procedure argument.

dummy_argument
can only appear once in the dummy argument list of any statement function. The dummy arguments have the scope of the statement function statement, and the same types and type parameters as the entities of the same names in the scoping unit containing the statement function.

Rules

A statement function is local to the scoping unit in which it is defined. It must not be defined in the scope of a module.

name determines the data type of the value returned from the statement function. If the data type of *name* does not match that of the scalar expression, the value of the scalar expression is converted to the type of *name* in accordance with the rules for assignment statements.

The names of the function and all the dummy arguments must be specified, explicitly or implicitly, to be scalar data objects.

The scalar expression can be composed of constants, references to variables, references to functions and function dummy procedures, and intrinsic operations. If the expression contains a reference to a function or function dummy procedure, the reference must not require an explicit interface, the function must not require an explicit interface or be a transformational intrinsic, and the result must be scalar. If an argument to a function or function dummy procedure is array-valued, it must be an array name.

With XL Fortran, the scalar expression can also reference a structure constructor.

The scalar expression can reference another statement function that is either:

- Declared previously in the same scoping unit, or
- Declared in the host scoping unit.

Named constants and arrays whose elements are referenced in the expression must be declared earlier in the scoping unit or be made accessible by use or host association.

Variables that are referenced in the expression must be either:
- Dummy arguments of the statement function, or
- Accessible in the scoping unit

If an entity in the expression is typed by the implicit typing rules, its type must agree with the type and type parameters given in any subsequent type declaration statement.

An external function reference in the scalar expression must not cause any dummy arguments of the statement function to become undefined or redefined.

If the statement function is defined in an internal subprogram and if it has the same name as an accessible entity from the host, precede the statement function definition with an explicit declaration of the statement function name. For example, use a type declaration statement.

The length specification for a statement function of type character or a statement function dummy argument of type character must be a constant specification expression.

Examples

```
PARAMETER (PI = 3.14159)
REAL AREA,CIRCUM,R,RADIUS
AREA(R) = PI * (R**2)            ! Define statement functions
CIRCUM(R) = 2 * PI * R           !   AREA and CIRCUM

! Reference the statement functions
PRINT *,'The area is: ',AREA(RADIUS)
PRINT *,'The circumference is: ',CIRCUM(RADIUS)
```

Related Information

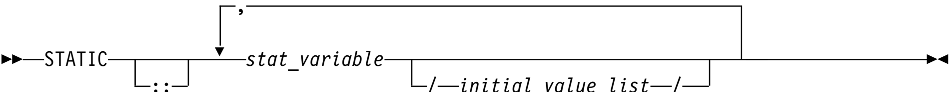
- “Dummy Arguments” on page 162
- “Function Reference” on page 159
- “How Type Is Determined” on page 52, for information on how the type of the statement function is determined

STATIC

Purpose

The **STATIC** attribute specifies that a variable has a storage class of static; that is, the variable remains in memory for the duration of the program and its value is retained between calls to the procedure.

Format



stat_variable
: is a variable name or an array declarator that can specify an *explicit_shape_spec_list* or a *deferred_shape_spec_list*.

initial_value
: provides an initial value for the variable specified by the immediately preceding name. Initialization occurs as described in "DATA" on page 277.

Rules

If *stat_variable* is a result variable, it must not be of type character or of derived type. Dummy arguments, automatic objects and pointees must not have the **STATIC** attribute. A variable that is explicitly declared with the **STATIC** attribute cannot be a common block item.

A variable must not have the **STATIC** attribute specified more than once in the same scoping unit.

Local variables have a default storage class of automatic. See the "**-qsave** Option" in the *User's Guide* for details on the default settings with regard to the invocation commands.

Variables declared as **STATIC** are shared amongst threads. To thread-safe an application that contains shared variables, you must either serialize access to the static data using locks, or make the data thread-specific. One method of making the data thread-specific is to move the static data into a **COMMON** block that has been declared **THREADLOCAL**. The **Pthreads** library module provides mutexes to allow you to serialize access to the data using locks. See "Chapter 15. Pthreads Library Module" on page 677 for more information. The *lock_name* attribute on the **CRITICAL** directive also provides the ability to serialize access to data. See "CRITICAL / END CRITICAL" on page 453 for

more information. The **THREADLOCAL** directive ensures that common blocks are local to each thread. See "THREADLOCAL" on page 507 for more information.

Attributes Compatible with the STATIC Attribute

- ALLOCATABLE
- DIMENSION
- POINTER
- PRIVATE
- SAVE
- TARGET
- VOLATILE

Examples

```
LOGICAL :: CALLED=.FALSE.
CALL SUB(CALLED)
CALLED=.TRUE.
CALL SUB(CALLED)
CONTAINS
  SUBROUTINE SUB(CALLED)
    INTEGER, STATIC :: J
    LOGICAL :: CALLED
    IF (CALLED.EQV..FALSE.) THEN
      J=2
    ELSE
      J=J+1
    ENDIF
    PRINT *, J                ! Output on first call is 2
                              ! Output on second call is 3
  END SUBROUTINE
END
```

Related Information

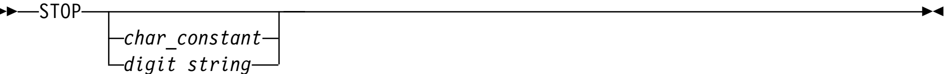
- "Storage Classes for Variables" on page 59
- "COMMON" on page 267
- "THREADLOCAL" on page 507

STOP

Purpose

When the **STOP** statement is executed, the program stops executing and, if a character constant or digit string is specified, prints the keyword **STOP** followed by the constant or digit string to unit 0.

Format



char_constant
is a scalar character constant that is not a Hollerith constant

digit_string
is a string of one through five digits

Rules

If neither *char_constant* nor *digit_string* are specified, nothing is printed to standard error (unit 0).

A **STOP** statement cannot terminate the range of a **DO** or **DO WHILE** construct.

If you specify *digit_string*, XL Fortran sets the system return code to **MOD** (*digit_string*,256). The system return code is available in the Korn shell command variable *$?*.

Examples

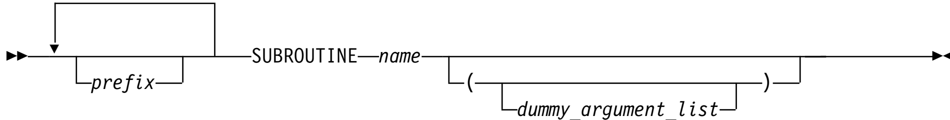
```
STOP 'Abnormal Termination'    ! Output:  STOP Abnormal Termination
END

STOP                           ! No output
END
```

SUBROUTINE

Purpose

The **SUBROUTINE** statement is the first statement of a subroutine subprogram.

Format



prefix is one of the following:
ELEMENTAL
PURE
RECURSIVE

Note: *type_spec* is not permitted as a prefix in a subroutine.

name is the name of the subroutine subprogram

Rules

At most one of each kind of *prefix* can be specified.

The subroutine name cannot appear in any other statement in the scope of the subroutine, unless recursion has been specified.

The **RECURSIVE** keyword must be specified if, directly or indirectly,
- The subroutine invokes itself.
- The subroutine invokes a procedure defined by an **ENTRY** statement in the same subprogram.
- An entry procedure in the same subprogram invokes itself.
- An entry procedure in the same subprogram invokes another entry procedure in the same subprogram.
- An entry procedure in the same subprogram invokes the subprogram defined by the **SUBROUTINE** statement.

If the **RECURSIVE** keyword is specified, the procedure interface is explicit within the subprogram.

Using the **PURE** or **ELEMENTAL** prefix indicates that the subroutine may be invoked by the compiler in any order as it is free of side effects. However, with regard to **PURE** subroutines, there are three exceptions:
- Dummy arguments with an intent of **OUT** or **INOUT** can be modified.
- The association status of dummy arguments with the **POINTER** attribute can be modified.
- The value of dummy arguments with the **POINTER** attribute can be modified.

You can also call external procedures recursively when you specify the **-qrecur** compiler option, although XL Fortran disregards this option if the **SUBROUTINE** statement specifies the **RECURSIVE** keyword.

For elemental procedures, the keyword **ELEMENTAL** must be specified. If the **ELEMENTAL** keyword is specified, the **RECURSIVE** keyword cannot be specified.

Examples

```
RECURSIVE SUBROUTINE SUB(X,Y)
  INTEGER X,Y
  IF (X.LT.Y) THEN
    RETURN
  ELSE
    CALL SUB(X,Y+1)
  END IF
END SUBROUTINE SUB
```

Related Information

- “Function and Subroutine Subprograms” on page 157
- “Dummy Arguments” on page 162
- “Recursion” on page 175
- “CALL” on page 257

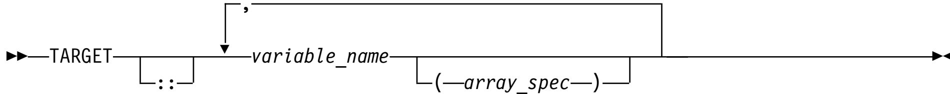
- "ENTRY" on page 303
- "RETURN" on page 385
- "Definition Status of Variables" on page 53
- "Pure Procedures" on page 176
- "**-qrecur** Option" in the *User's Guide*

TARGET

Purpose

Variables with the **TARGET** attribute can become pointer targets.

Format



Rules

Although the target of a pointer can also be a pointer, this target cannot have the **TARGET** attribute.

A target cannot appear in an **EQUIVALENCE** statement.

A target cannot be an integer pointer or a pointee.

Attributes Compatible with the TARGET Attribute

- ALLOCATABLE
- AUTOMATIC
- DIMENSION
- INTENT
- OPTIONAL
- PRIVATE
- PUBLIC
- SAVE
- STATIC
- VOLATILE

Examples

```
REAL, POINTER :: A,B
REAL, TARGET  :: C = 3.14
B => C
A => B       ! A points to C
```

Related Information

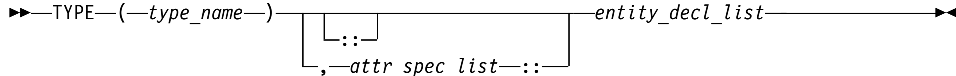
- "POINTER (Fortran 90)" on page 364
- "ALLOCATED(ARRAY)" on page 529
- "DEALLOCATE" on page 280
- "Pointer Assignment" on page 117
- "Pointer Association" on page 139

TYPE

Purpose

A **TYPE** type declaration statement specifies the type and attributes of objects and functions of derived type. Initial values can be assigned to objects.

Format



where:

attr_spec
ALLOCATABLE
AUTOMATIC
DIMENSION (*array_spec*)
EXTERNAL
INTENT (*intent_spec*)
INTRINSIC
OPTIONAL
PARAMETER
POINTER
PRIVATE
PUBLIC
SAVE
STATIC
TARGET
VOLATILE

type_name
is the name of a derived type

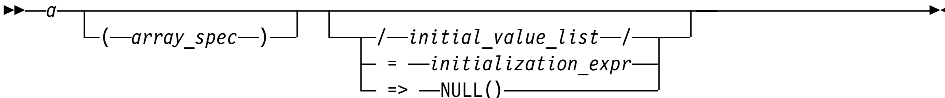
attr_spec
For detailed information on rules about a particular attribute, refer to the statement of the same name.

intent_spec
is either **IN**, **OUT**, or **INOUT**

:: is the double colon separator. It is required if attributes are specified, = *initialization_expr* is used, or =>**NULL()** appears as part of any *entity_decl*.

array_spec
is a list of dimension bounds

entity_decl



a is an object name or function name. *array_spec* cannot be specified for a function name.

initial_value
provides an initial value for the entity specified by the immediately preceding name. Initialization occurs as described in “DATA” on page 277.

initialization_expr
provides an initial value, by means of an initialization expression, for the entity specified by the immediately preceding name

=> **NULL()**
provides the initial value for a pointer object

Rules

Within the context of a derived type definition:

- If => appears in a component initialization, the **POINTER** attribute must appear in the *attr_spec_list*.
- If = appears in a component initialization, the **POINTER** attribute cannot appear in the component *attr_spec_list*.
- The compiler will evaluate *initialization_expr* within the scoping unit of the type definition.

If => appears for a variable, the object must have the **POINTER** attribute.

If *initialization_expr* appears for a variable, the object cannot have the **POINTER** attribute.

Entities in type declaration statements are constrained by the rules of any attributes specified for the entities, as detailed in the corresponding attribute statements.

Once a derived type has been defined, you can use it to define your data items using the **TYPE** type declaration statement. When an entity is explicitly declared to be of a derived type, that derived type must have been previously defined in the scoping unit or is accessible by use or host association.

The data object becomes an *object of derived type* or a *structure*. Each *structure component* is a subobject of the object of derived type.

If you specify the **DIMENSION** attribute, you are creating an array whose elements have a data type of that derived type.

Other than in specification statements, you can use objects of derived type as actual and dummy arguments, and they can also appear as items in input/output lists (unless the object has a component with the **POINTER** attribute), assignment statements, structure constructors, and the right side of a statement function definition. If a structure component is not accessible, a derived-type object cannot be used in an input/output list or as a structure constructor.

Objects of nonsequence derived type cannot be used as data items in **EQUIVALENCE** and **COMMON** statements. Objects of nonsequence data types cannot be integer pointees.

A nonsequence derived-type dummy argument must specify a derived type that is accessible through use or host association to ensure that the same derived-type definition defines both the actual and dummy arguments.

The type declaration statement overrides the implicit type rules in effect.

An object cannot be initialized in a type declaration statement if it is a dummy argument, allocatable array, pointer, function result, object in blank common, integer pointer, external name, intrinsic name, or automatic object. Nor can an object be initialized if it has the **AUTOMATIC** attribute. The object may be initialized if it appears in a named common block in a block data program unit or if it appears in a named common block in a module.

In Fortran 95, a pointer can be initialized. Pointers can only be initialized by the use of **=> NULL()**.

The specification expression of an *array_spec* can be a nonconstant expression if the specification expression appears in an interface body or in the specification part of a subprogram. Any object being declared that uses this nonconstant expression and is not a dummy argument or a pointee is called an *automatic object*.

An attribute cannot be repeated in a given type declaration statement, nor can an entity be explicitly given the same attribute more than once in a scoping unit.

initialization_expr must be specified if the statement contains the **PARAMETER** attribute. If the entity you are declaring is a variable, and *initialization_expr* or **NULL()** is specified, the variable is initially defined. If the entity you are declaring is a derived type component, and *initialization_expr* or **NULL()** is specified, the derived type has default initialization. *a* becomes defined with

the value determined by *initialization_expr*, in accordance with the rules for intrinsic assignment. If the entity is an array, its shape must be specified either in the type declaration statement or in a previous specification statement in the same scoping unit. A variable or variable subobject cannot be initialized more than once. If *a* is a variable, the presence of *initialization_expr* or =>**NULL()** implies that *a* is a saved object, except for an object in a named common block. The initialization of an object could affect the fundamental storage class of an object.

An *array_spec* specified in the *entity_decl* takes precedence over the *array_spec* in the **DIMENSION** attribute.

An array function result that does not have the **POINTER** attribute must have an explicit-shape array specification.

If the entity declared is a function, it must not have an accessible explicit interface unless it is an intrinsic function. The derived type can be specified on the **FUNCTION** statement, provided the derived type is defined within the body of the function or is accessible via host or use association.

If `T` or `F`, defined previously as the name of a constant, appears in a type declaration statement, it is no longer an abbreviated logical constant but the name of the named constant.

Examples

```
TYPE PEOPLE                       ! Defining derived type PEOPLE
  INTEGER AGE
  CHARACTER*20 NAME
END TYPE PEOPLE
TYPE(PEOPLE) :: SMITH = PEOPLE(25,'John Smith')
END
```

Related Information

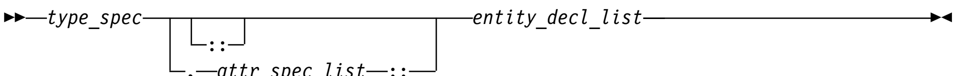
- “Derived Types” on page 39
- “Derived Type” on page 282
- “Initialization Expressions” on page 87
- “How Type Is Determined” on page 52, for details on the implicit typing rules
- “Array Declarators” on page 65
- “Automatic Objects” on page 29
- “Storage Classes for Variables” on page 59

Type Declaration

Purpose

A type declaration statement specifies the type, length, and attributes of objects and functions. Initial values can be assigned to objects.

Format

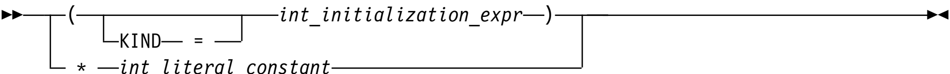


where:

type_spec	*attr_spec*
BYTE **CHARACTER** [*char_selector*] **COMPLEX** [*kind_selector*] **DOUBLE COMPLEX** **DOUBLE PRECISION** **INTEGER** [*kind_selector*] **LOGICAL** [*kind_selector*] **REAL** [*kind_selector*] **TYPE** (*type_name*)	**ALLOCATABLE** **AUTOMATIC** **DIMENSION** (*array_spec*) **EXTERNAL** **INTENT** (*intent_spec*) **INTRINSIC** **OPTIONAL** **PARAMETER** **POINTER** **PRIVATE** **PUBLIC** **SAVE** **STATIC** **TARGET** **VOLATILE**

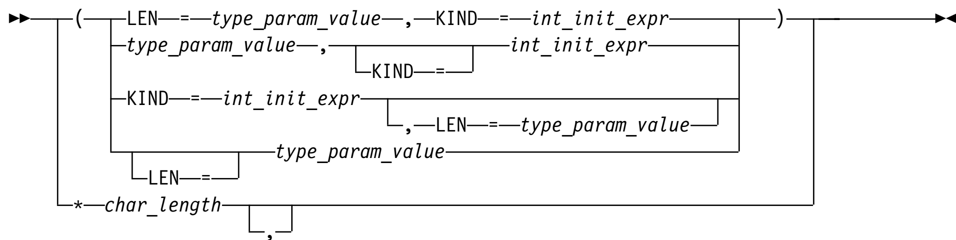
type_name
is the name of a derived type

kind_selector



represents one of the permissible length specifications for its associated type.*int_literal_constant* cannot specify a kind type parameter.

char_selector
specifies the character length(number of characters between 0 and 256 MB). Values exceeding 256 MB are set to 256 MB, while negative values result in a length of zero. If not specified, the default length is 1. The kind type parameter, if specified, must be 1, which specifies the ASCII character representation.



type_param_value
: is a specification expression or an asterisk (*)

int_init_expr
: is a scalar integer initialization expression that must evaluate to 1

char_length
: is either a scalar integer literal constant (which cannot specify a kind type parameter) or a *type_param_value* enclosed in parentheses

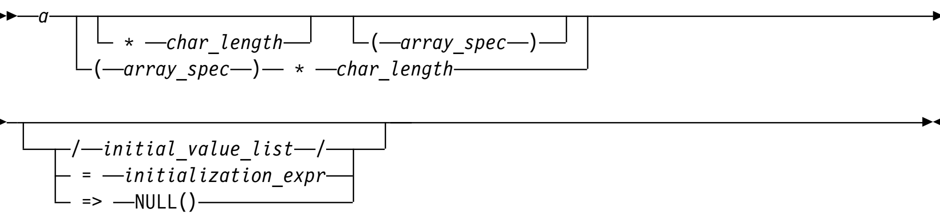
attr_spec
: For detailed information on rules about a particular attribute, refer to the statement of the same name.

intent_spec
: is either **IN**, **OUT**, or **INOUT**

::
: is the double colon separator. It is required if attributes are specified,= *initialization_expr* or => **NULL()** is used

array_spec
: is a list of dimension bounds

entity_decl



a
: is an object name or function name. *array_spec* cannot be specified for a function name.

char_length
overrides the length as specified in *kind_selector* and *char_selector*, and is only permitted in statements where the length can be specified with the initial keyword. A character entity can specify *char_length*, as defined above. A noncharacter entity can only specify an integer literal constant that represents one of the permissible length specifications for its associated type.

initial_value
provides an initial value for the entity specified by the immediately preceding name.

initialization_expr
provides an initial value, by mean of an initialization expression, for the entity specified by the immediately preceding name.

=> **NULL()**
provides the initial value for the pointer object.

Rules

Within the context of a derived type definition:

- If => appears in a component initialization, the **POINTER** attribute must appear in the *attr_spec_list*.
- If = appears in a component initialization, the **POINTER** attribute cannot appear in the component *attr_spec_list*.
- The compiler will evaluate *initialization_expr* within the scoping unit of the type definition.

If => appears for a variable, the object must have the **POINTER** attribute.

If *initialization_expr* appears for a variable, the object cannot have the **POINTER** attribute.

Entities in type declaration statements are constrained by the rules of any attributes specified for the entities, as detailed in the corresponding attribute statements.

The type declaration statement overrides the implicit type rules in effect. You can use a type declaration statement that confirms the type of an intrinsic function. The appearance of a generic or specific intrinsic function name in a type declaration statement does not cause the name to lose its intrinsic property.

An object cannot be initialized in a type declaration statement if it is a dummy argument, allocatable array, pointer, function result, object in blank

common, integer pointer, external name, intrinsic name, or automatic object. Nor can an object be initialized if it has the **AUTOMATIC** attribute. The object may be initialized if it appears in a named common block in a block data program unit or if it appears in a named common block in a module.

In Fortran 95, a pointer can be initialized. Pointers can only be initialized by the use of **=> NULL()**.

The specification expression of a *type_param_value* or an *array_spec* can be a nonconstant expression if the specification expression appears in an interface body or in the specification part of a subprogram. Any object being declared that uses this nonconstant expression and is not a dummy argument or a pointee is called an *automatic object*.

An attribute cannot be repeated in a given type declaration statement, nor can an entity be explicitly given the same attribute more than once in a scoping unit.

initialization_expr must be specified if the statement contains the **PARAMETER** attribute. If the entity you are declaring is a variable, and *initialization_expr* or **NULL()** is specified, the variable is initially defined. If the entity you are declaring is a derived type component, and *initialization_expr* or **NULL()** is specified, the derived type has default initialization. *a* becomes defined with the value determined by *initialization_expr*, in accordance with the rules for intrinsic assignment. If the entity is an array, its shape must be specified either in the type declaration statement or in a previous specification statement in the same scoping unit. A variable or variable subobject cannot be initialized more than once. If *a* is a variable, the presence of *initialization_expr* or =>**NULL()** implies that *a* is a saved object, except for an object in a named common block. The initialization of an object could affect the fundamental storage class of an object.

An *array_spec* specified in an *entity_decl* takes precedence over the *array_spec* in the **DIMENSION** attribute.

An array function result that does not have the **POINTER** attribute must have an explicit-shape array specification.

If the entity declared is a function, it must not have an accessible explicit interface unless it is an intrinsic function.

If `T` or `F`, defined previously as the name of a constant, appears in a type declaration statement, it is no longer an abbreviated logical constant but the name of the named constant.

The optional comma after *char_length* in a **CHARACTER** type declaration statement is permitted only if no double colon separator (::) appears in the statement.

If the **CHARACTER** type declaration statement is in the scope of a module, block data program unit, or main program, and you specify the length of the entity as an inherited length, the entity must be the name of a named character constant. The character constant assumes the length of its corresponding expression defined by the **PARAMETER** attribute.

If the **CHARACTER** type declaration statement is in the scope of a procedure and the length of the entity is inherited, the entity name must be the name of a dummy argument or a named character constant. If the statement is in the scope of an external function, it can also be the function or entry name in a **FUNCTION** or **ENTRY** statement in the same program unit. If the entity name is the name of a dummy argument, the dummy argument assumes the length of the associated actual argument for each reference to the procedure. If the entity name is the name of a character constant, the character constant assumes the length of its corresponding expression defined by the **PARAMETER** attribute. If the entity name is a function or entry name, the entity assumes the length specified in the calling scoping unit.

The length of a character function is either a specification expression (which must be a constant expression if the function type is not declared in an interface block) or it is an asterisk, indicating the length of a dummy procedure name. The length cannot be an asterisk if the function is an internal or module function, if it is recursive, or if it returns array or pointer values.

Examples

```
CHARACTER(KIND=1,LEN=6) APPLES /'APPLES'/
CHARACTER*7, TARGET :: ORANGES = 'ORANGES'
CALL TEST(APPLES)
END

SUBROUTINE  TEST(VARBL)
  CHARACTER*(*), OPTIONAL :: VARBL   ! VARBL inherits a length of 6

  COMPLEX, DIMENSION (2,3) :: ABC(3) ! ABC has 3 (not 6) array elements
  REAL, POINTER :: XCONST

  TYPE PEOPLE                        ! Defining derived type PEOPLE
    INTEGER AGE
    CHARACTER*20 NAME
  END TYPE PEOPLE
  TYPE(PEOPLE) :: SMITH = PEOPLE(25,'John Smith')
END
```

Related Information

- "Chapter 3. Data Types and Data Objects" on page 27
- "Initialization Expressions" on page 87
- "How Type Is Determined" on page 52, for details on the implicit typing rules
- "Array Declarators" on page 65
- "Automatic Objects" on page 29
- "Storage Classes for Variables" on page 59
- "DATA" on page 277, for details on initial values

USE

Purpose

The **USE** statement is a module reference that provides local access to the public entities of a module.

Format

rename is the assignment of a local name to an accessible data entity:
`local_name => use_name`

only is a *rename*, a generic specification, or the name of a variable, procedure, derived type, named constant, or namelist group

Rules

The **USE** statement can only appear prior to all other statements in *specification_part*. Multiple **USE** statements may appear within a scoping unit.

At the time the file containing the **USE** statement is being compiled, the specified module must precede the **USE** statement in the file or the module must have been already compiled in another file. Each referenced entity must be the name of a public entity in the module.

Entities in the scoping unit become *use-associated* with the module entities, and the local entities have the attributes of the corresponding module entities.

In addition to the **PRIVATE** attribute, the **ONLY** clause of the **USE** statement provides further constraint on which module entities can be accessed. If the **ONLY** clause is specified, only entities named in the *only_list* are accessible. If no list follows the keyword, no module entities are accessible. If the **ONLY** clause is absent, all public entities are accessible.

If a scoping unit contains multiple **USE** statements, all specifying the same module, and one of the statements does not include the **ONLY** clause, all public entities are accessible. If each **USE** statement includes the **ONLY** clause, only those entities named in one or more of the *only_lists* are accessible.

You can rename an accessible entity for local use. A module entity can be accessed by more than one local name. If no renaming is specified, the name of the use-associated entity becomes the local name. The local name of a use-associated entity cannot be redeclared. However, if the **USE** statement appears in the scoping unit of a module, the local name can appear in a **PUBLIC** or **PRIVATE** statement.

If multiple generic interfaces that are accessible to a scoping unit have the same local name, operator, or assignment, they are treated as a single generic interface. In such a case, one of the generic interfaces can contain an interface body to an accessible procedure with the same name. Otherwise, any two different use-associated entities can only have the same name if the name is not used to refer to an entity in the scoping unit. If a use-associated entity and host entity share the same name, the host entity becomes inaccessible through host association by that name.

A module must not reference itself, either directly or indirectly. For example, module X cannot reference module Y if module Y references module X.

Consider the situation where a module (for example, module B) has access through use association to the public entities of another module (for example, module A). The accessibility of module B's local entities (which includes those entities that are use-associated with entities from module A) to other program units is determined by the **PRIVATE** and **PUBLIC** attributes, or, if absent, through the default accessibility of module B. Of course, other program units can access the public entities of module A directly.

Examples

```
MODULE A
  REAL :: X=5.0
END MODULE A
MODULE B
  USE A
  PRIVATE :: X               !  X cannot be accessed through module B
  REAL :: C=80, D=50
END MODULE B
PROGRAM TEST
  INTEGER :: TX=7
  CALL SUB
  CONTAINS

  SUBROUTINE SUB
  USE B, ONLY : C
  USE B, T1 => C
```

```
    USE B, TX => C           !  C is given another local name
    USE A
    PRINT *, TX              !  Value written is 80 because use-associated
                             !  entity overrides host entity
    END SUBROUTINE
  END
```

Related Information

- "Modules" on page 153
- "PRIVATE" on page 370
- "PUBLIC" on page 373
- "Order of Statements and Execution Sequence" on page 25

VIRTUAL

Purpose

The **VIRTUAL** statement specifies the name and dimensions of an array. It is an alternative form of the **DIMENSION** statement, although there is no **VIRTUAL** attribute.

Format

```
>>—VIRTUAL—array_declarator_list——————————————————————————————————><
```

Rules

You can specify arrays with a maximum of 20 dimensions.

Only one array specification for an array name can appear in a scoping unit.

Examples

```
VIRTUAL A(10), ARRAY(5,5,5), LIST(10,100)
VIRTUAL ARRAY2(1:5,1:5,1:5), LIST2(I,M)       ! adjustable array
VIRTUAL B(0:24), C(-4:2), DATA(0:9,-5:4,10)
VIRTUAL ARRAY (M*N*J,*)                       ! assumed-size array
```

Related Information

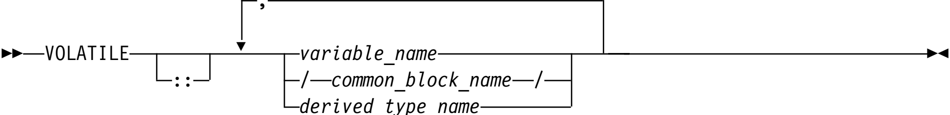
- "Chapter 4. Array Concepts" on page 63
- "DIMENSION" on page 283

VOLATILE

Purpose

The **VOLATILE** attribute is used to designate a data object as being mapped to memory that can be accessed by independent input/output processes and independent, asynchronously interrupting processes. Code that manipulates volatile data objects is not optimized.

Format



Rules

If an array name is declared volatile, each element of the array is considered volatile. If a common block is declared volatile, each variable in the common block is considered volatile. An element of a common block can be declared volatile without affecting the status of the other elements in the common block.

If a common block is declared in multiple scopes, and if it (or one or more of its elements) is declared volatile in one of those scopes, you must specify the **VOLATILE** attribute in each scope where you require the common block (or one or more of its elements) to be considered volatile.

If a derived type name is declared volatile, all variables declared with that type are considered volatile. If an object of derived type is declared volatile, all of its components are considered volatile. If a component of a derived type is itself derived, the component does not inherit the volatile attribute from its type. A derived type name that is declared volatile must have had the **VOLATILE** attribute prior to any use of the type name in a type declaration statement.

If a pointer is declared volatile, the storage of the pointer itself is considered volatile. The **VOLATILE** attribute has no effect on any associated pointer targets.

If you declare an object to be volatile and then use it in an **EQUIVALENCE** statement, all of the objects that are associated with the volatile object through equivalence association are considered volatile.

Any data object that is shared across threads and is stored and read by multiple threads must be declared as **VOLATILE**. If, however, your program only uses the automatic or directive-based parallelization facilities of the compiler, variables that have the **SHARED** attribute need not be declared **VOLATILE**.

If the actual argument associated with a dummy argument is a variable that is declared volatile, you must declare the dummy argument volatile if you require the dummy argument to be considered volatile. If a dummy argument

is declared volatile, and you require the associated actual argument to be considered volatile, you must declare the actual argument as volatile.

Declaring a statement function as volatile has no effect on the statement function.

Within a function subprogram, the function result variable can be declared volatile. Any entry result variables will be considered volatile. An **ENTRY** name must not be specified with the **VOLATILE** attribute.

Attributes Compatible with the VOLATILE Attribute

- ALLOCATABLE
- AUTOMATIC
- DIMENSION
- INTENT
- OPTIONAL
- POINTER
- PRIVATE
- PUBLIC
- SAVE
- STATIC
- TARGET

Examples

```
      FUNCTION TEST ()
        REAL ONE, TWO, THREE
        COMMON /BLOCK1/A, B, C
        ...
        VOLATILE /BLOCK1/, ONE, TEST
! Common block elements A, B and C are considered volatile
! since common block BLOCK1 is declared volatile.
        ...
        EQUIVALENCE (ONE, TWO), (TWO, THREE)
! Variables TWO and THREE are volatile as they are equivalenced
! with variable ONE which is declared volatile.
      END FUNCTION
```

Related Information

"Chapter 11. Directives" on page 423

WAIT

Purpose

The **WAIT** statement may be used to wait for an asynchronous data transfer to complete or it may be used to detect the completion status of an asynchronous data transfer statement.

Format

```
►►—WAIT—(—wait_list—)————————————————————————————————————————►◄
```

wait_list
is a list that must contain one **ID=** specifier and at most one of each of the other valid specifiers. The valid specifiers are:

ID= *integer_expr*
indicates the data transfer with which this **WAIT** statement is identified. The *integer_expr* is a scalar of type **INTEGER(4)** or default integer. To initiate an asynchronous data transfer, the **ID=** specifier is used on a **READ** or **WRITE** statement.

IOSTAT= *ios*
is an input/output status specifier that specifies the status of the input/output operation. *ios* is a scalar variable of type **INTEGER(4)** or default integer. When the input/output statement containing this specifier finishes execution, *ios* is defined with:
- A zero value if no error condition occurs;
- A positive value if an error occurs;
- A negative value if an end-of-file condition is encountered and no error occurs.

The *ios* defined for the **IOSTAT=** specifier of the asynchronous data transfer statement need not be identical to the *ios* defined for the **IOSTAT=** specifier of the matching **WAIT** statement.

ERR= *stmt_label*
is an error specifier that specifies the statement label of an executable statement in the same scoping unit to which control is to transfer in case of an error. Coding the **ERR=** specifier suppresses error messages.

The *stmt_label* defined for the **ERR=** specifier of the asynchronous data transfer statement need not be identical to the *stmt_label* defined for the **ERR=** specifier of the matching **WAIT** statement.

END= *stmt_label*
is an end-of-file specifier that specifies a statement label at which the program is to continue if an endfile record is encountered and no error occurs. If an external file is positioned after the endfile record, the **IOSTAT=** specifier, if present, is assigned a negative value, and the **NUM=** specifier, if present, is assigned an integer value. Coding the **END=** specifier suppresses the error message for end-of-file. This specifier can be specified for a unit connected for either sequential or direct access.

The *stmt_label* defined for the **END=** specifier of the asynchronous data transfer statement need not be identical to the *stmt_label* defined for the **END=** specifier of the matching **WAIT** statement.

DONE= *logical_variable*
specifies whether or not the asynchronous I/O statement is complete.

If the **DONE=** specifier is present, the *logical_variable* is set to true if the asynchronous I/O is complete and is set to false if it is not complete. If the returned value is false, then one or more **WAIT** statements must be executed until either the **DONE=** specifier is not present, or its returned value is true. A **WAIT** statement without the **DONE=** specifier, or a **WAIT** statement that sets the *logical_variable* value to true, is the matching **WAIT** statement to the data transfer statement identified by the same **ID=** value.

Rules

The matching **WAIT** statement must be in the same scoping unit as the corresponding asynchronous data transfer statement. Within the instance of that scoping unit, the program must not execute a **RETURN**, **END**, or **STOP** statement before the matching **WAIT** statement is executed.

Related Information

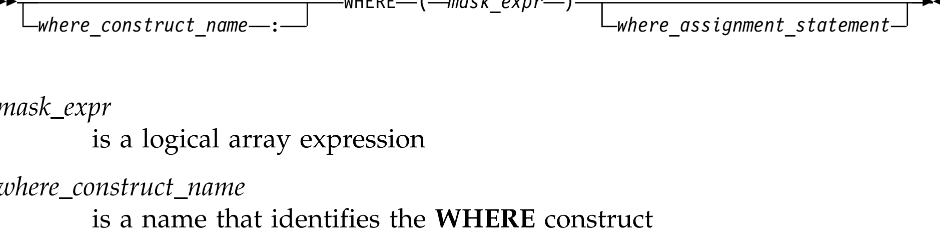
"Executing Data Transfer Statements Asynchronously" on page 189
"AIX Implementation Details of XL Fortran Input/Output" in the *User's Guide*

WHERE

Purpose

The **WHERE** statement masks the evaluation of expressions and assignments of values in array assignment statements. It does this according to the value of a logical array expression. The **WHERE** statement can be the initial statement of the **WHERE** construct.

Format



mask_expr
is a logical array expression

where_construct_name
is a name that identifies the **WHERE** construct

Rules

If a *where_assignment_statement* is present, the **WHERE** statement is not the first statement of a **WHERE** construct. If a *where_assignment_statement* is absent, the **WHERE** statement is the first statement of the **WHERE** construct, and is referred to as a **WHERE** construct statement. An **END WHERE** statement must follow. See "WHERE Construct" on page 106 for more information.

If the **WHERE** statement is not the first statement of a **WHERE** construct, you can use it as the terminal statement of a **DO** or **DO WHILE** construct.

You can nest **WHERE** statements within a **WHERE** construct. A *where_assignment_statement* that is a defined assignment must be an elemental defined assignment.

In each *where_assignment_statement*, the *mask_expr* and the *variable* being defined must be arrays of the same shape. Each *mask_expr* in a **WHERE** construct must have the same shape. A **WHERE** statement that is part of a *where_body_construct* must not be a branch target statement. The execution of a function reference in the *mask_expr* of a **WHERE** statement can affect entities in the *where_assignment_statement*.

See “Interpreting Masked Array Assignments” on page 108 for information on interpreting mask expressions.

If a *where_construct_name* appears on a **WHERE** construct statement, it must also appear on the corresponding **END WHERE** statement. A construct name is optional on any masked **ELSEWHERE** and **ELSEWHERE** statements in the **WHERE** construct.

A *where_construct_name* can only appear on a **WHERE** construct statement.

Examples

```
REAL, DIMENSION(10) :: A,B,C

!   In the following WHERE statement, the LOG of an element of A
!   is assigned to the corresponding element of B only if that
!   element of A is a positive value.

WHERE (A>0.0) B = LOG(A)

   :

END
```

The following example shows an elemental defined assignment in a **WHERE** statement:

```
INTERFACE ASSIGNMENT(=)
  ELEMENTAL SUBROUTINE MY_ASSIGNMENT(X, Y)
    LOGICAL, INTENT(OUT) :: X
    REAL, INTENT(IN) :: Y
  END SUBROUTINE MY_ASSIGNMENT
END INTERFACE

INTEGER A(10)
REAL C(10)
LOGICAL L_ARR(10)
```

```
C = (/ -10., 15.2, 25.5, -37.8, 274.8, 1.1, -37.8, -36.2, 140.1, 127.4 /)
A = (/ 1, 2, 7, 8, 3, 4, 9, 10, 5, 6 /)
L_ARR = .FALSE.

WHERE (A < 5) L_ARR = C

! DATA IN ARRAY L_ARR AT THIS POINT:
!
! L_ARR = F, T, F, F, T, T, F, F, F, F

END

ELEMENTAL SUBROUTINE MY_ASSIGNMENT(X, Y)
  LOGICAL, INTENT(OUT) :: X
  REAL, INTENT(IN) :: Y

  IF (Y < 0.0) THEN
    X = .FALSE.
  ELSE
    X = .TRUE.
  ENDIF
END SUBROUTINE MY_ASSIGNMENT
```

Related Information

- "WHERE Construct" on page 106
- "ELSEWHERE" on page 295
- "END (Construct)" on page 298, for details on the **END WHERE** statement

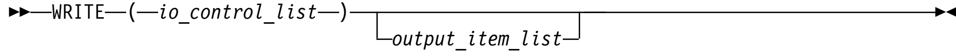
---

WRITE

Purpose

The **WRITE** statement is a data transfer output statement.

Format



output_item
is an output list item. An output list specifies the data to be transferred. An output list item can be:
- A variable name. An array is treated as if all of its elements were specified in the order in which they are arranged in storage.

  A pointer must be associated with a target, and an allocatable array must be allocated. A derived-type object cannot have any ultimate component that is outside the scoping unit of this statement. The evaluation of *output_item* cannot result in a derived-type object that contains a pointer. The structure components of a structure in a

formatted statement are treated as if they appear in the order of the derived-type definition; in an unformatted statement, the structure components are treated as a single value in their internal representation (including padding).

- An expression
- An implied-**DO** list, as described under "Implied-DO List" on page 420

io_control

is a list that must contain one unit specifier (**UNIT=**), and can also contain one of each of the other valid specifiers:

[UNIT=] *u*

is a unit specifier that specifies the unit to be used in the output operation. *u* is an external unit identifier or internal file identifier.

An external unit identifier refers to an external file. It is one of the following:

- An integer expression whose value is in the range 0 through 2,147,483,647.
- An asterisk, which identifies external unit 6 and is preconnected to standard output.

An internal file identifier refers to an internal file. It is the name of a character variable, which cannot be an array section with a vector subscript.

If the optional characters **UNIT=** are omitted, *u* must be the first item in *io_control_list*. If **UNIT=** is specified, **FMT=** must also be specified.

[FMT=] *format*

is a format specifier that specifies the format to be used in the output operation. *format* is a format identifier that can be:

- The statement label of a **FORMAT** statement. The **FORMAT** statement must be in the same scoping unit.
- The name of a scalar **INTEGER(4)** or **INTEGER(8)** variable that was assigned the statement label of a **FORMAT** statement. The **FORMAT** statement must be in the same scoping unit.

  Fortran 95 does not permit assigning of a statement label.
- A character constant enclosed in parentheses. Only the format codes listed under "FORMAT" on page 315 can be used between the parentheses. Blank characters can precede the left parenthesis or follow the right parenthesis.
- A character variable that contains character data whose leftmost character positions constitute a valid format. A valid format begins with a left parenthesis and ends with a right parenthesis. Only the format codes described in the **FORMAT** statement can be used

between the parentheses. Blank characters can precede the left parenthesis or follow the right parenthesis. If *format* is an array element, the format identifier must not exceed the length of the array element.

- An array of noncharacter intrinsic type. The data must be a valid format identifier as described under character array.
- Any character expression, except one involving concatenation of an operand that specifies inherited length, unless the operand is the name of a constant.
- An asterisk, specifying list-directed formatting.
- A namelist specifier that specifies the name of a namelist list that you have previously defined.

If the optional characters **FMT=** are omitted, *format* must be the second item in *io_control_list*, and the first item must be the unit specifier with **UNIT=** omitted. **NML=** and **FMT=** cannot both be specified in the same output statement.

REC= *integer_expr*
is a record specifier that specifies the number of the record to be written in a file connected for direct access. The **REC=** specifier is only permitted for direct output. *integer_expr* is an integer expression whose value is positive. A record specifier is not valid if formatting is list-directed or if the unit specifier specifies an internal file. The record specifier represents the relative position of a record within a file. The relative position number of the first record is 1.

IOSTAT= *ios*
is an input/output status specifier that specifies the status of the input/output operation. *ios* is a scalar variable of type **INTEGER(4)** or default integer. Coding the **IOSTAT=** specifier suppresses error messages. When the statement finishes execution, *ios* is defined with:
- A zero value if no error condition occurs
- A positive value if an error occurs.

ID= *integer_variable*
indicates that the data transfer is to be done asynchronously. The *integer_variable* is a scalar of type **INTEGER(4)** or default integer. If no error is encountered, the *integer_variable* is defined with a value after executing the asynchronous data transfer statement. This value must be used in the matching **WAIT** statement.

Asynchronous data transfer must either be direct unformatted or sequential unformatted. Asynchronous I/O to internal files is prohibited. Asynchronous I/O to raw character devices (for example, tapes or raw logical volumes) is prohibited. The *integer_variable* must not be associated with any entity in the data transfer I/O list, or with

a *do_variable* of an *io_implied_do* in the data transfer I/O list. If the *integer_variable* is an array element reference, its subscript values must not be affected by the data transfer, the *io_implied_do* processing, or the definition or evaluation of any other specifier in the *io_control_spec*.

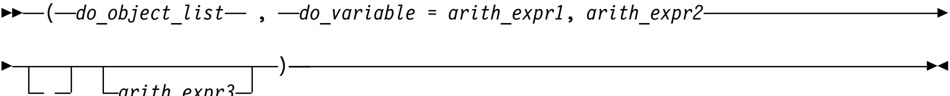
ERR= *stmt_label*
: is an error specifier that specifies the statement label of an executable statement in the same scoping unit to which control is to transfer in the case of an error. Coding the **ERR=** specifier suppresses error messages.

NUM= *integer_variable*
: is a number specifier that specifies the number of bytes of data transmitted between the I/O list and the file. *integer_variable* is a variable name of type **INTEGER(4)**, type **INTEGER(8)** in 64-bit, or type default integer. The **NUM=** specifier is only permitted for unformatted output. Coding the **NUM** parameter suppresses the indication of an error that would occur if the number of bytes represented by the output list is greater than the number of bytes that can be written into the record. In this case, *integer_variable* is set to a value that is the maximum length record that can be written. Data from remaining output list items is not written into subsequent records. In the portion of the program that executes between the asynchronous data transfer statement and the matching **WAIT** statement, the *integer_variable* in the **NUM=** specifier or any variable associated with it must not be referenced, become defined, or become undefined.

[NML=] *name*
: is a namelist specifier that specifies the name of a namelist list that you have previously defined. If the optional characters **NML=** are not specified, the namelist name must appear as the second parameter in the list, and the first item must be the unit specifier with **UNIT=** omitted. If both **NML=** and **UNIT=** are specified, all the parameters can appear in any order. The **NML=** specifier is an alternative to **FMT=**. Both **NML=** and **FMT=** cannot be specified in the same output statement.

ADVANCE= *char_expr*
: is an advance specifier that determines whether nonadvancing output occurs for this statement. *char_expr* is a character expression that must evaluate to **YES** or **NO**. If **NO** is specified, nonadvancing output occurs. If **YES** is specified, advancing, formatted sequential output occurs. The default value is **YES**. **ADVANCE=** can be specified only in a formatted sequential **WRITE** statement with an explicit format specification that does not specify an internal file unit specifier.

Implied-DO List



do_object
: is an output list item

do_variable
: is a named scalar variable of type integer or real

arith_expr1, *arith_expr2*, and *arith_expr3*
: are scalar numeric expressions

The range of an implied-**DO** list is the list *do_object_list*. The iteration count and values of the **DO** variable are established from *arith_expr1*, *arith_expr2*, and *arith_expr3*, the same as for a **DO** statement. When the implied-**DO** list is executed, the items in the *do_object_list* are specified once for each iteration of the implied-**DO** list, with the appropriate substitution of values for any occurrence of the **DO** variable.

Rules

If a **NUM=** specifier is present, neither a format specifier nor a namelist specifier can be present.

Variables specified for the **IOSTAT=** and **NUM=** specifiers must not be associated with any output list item, namelist list item, or **DO** variable of an implied-**DO** list. If such a specifier variable is an array element, its subscript values must not be affected by the data transfer, any implied-**DO** processing, or the definition or evaluation of any other specifier.

If the **ERR=** and **IOSTAT=** specifiers are set and an error is encountered during a synchronous data transfer, transfer is made to the statement specified by the **ERR=** specifier and a positive integer value is assigned to *ios*.

If the **ERR=** or **IOSTAT=** specifiers are set and an error is encountered during an asynchronous data transfer, execution of the matching **WAIT** statement is not required.

If a conversion error is encountered and the **CNVERR** run-time option is set to **NO**, **ERR=** is not branched to, although **IOSTAT=** may be set.

If **IOSTAT=** and **ERR=** are not specified,
- The program stops if a severe error is encountered.

- The program continues to the next statement if a recoverable error is encountered and the **ERR_RECOVERY** run-time option is set to **YES**. If the option is set to **NO**, the program stops.
- The program continues to the next statement when a conversion error is encountered if the **ERR_RECOVERY** run-time option is set to **YES**. If the **CNVERR** run-time option is set to **YES**, conversion errors are treated as recoverable errors; when **CNVERR=NO**, they are treated as conversion errors.

`PRINT format` has the same effect as `WRITE(*,format)`.

Examples

```
WRITE (6,FMT='(10F8.2)') (LOG(A(I)),I=1,N+9,K),G
```

Related Information

- “Executing Data Transfer Statements Asynchronously” on page 189
- ″AIX Implementation Details of XL Fortran Input/Output″ in the *User's Guide*
- “Conditions and IOSTAT Values” on page 193
- “Chapter 8. Input/Output Concepts” on page 183
- “READ” on page 374
- “WAIT” on page 412
- ″Setting Runtime Options for Input/Output″ in the *User's Guide*
- “Deleted Features” on page 735

Chapter 11. Directives

This chapter provides an alphabetical reference to the compiler directives and includes background information on the following:

- The XLF directives used for SMP programming. To ensure that your compiler will recognize these directives, compile your code using either the **xlf_r**, **xlf_r7**, **xlf90_r**, **xlf90_r7**, **xlf95_r**, or **xlf95_r7** invocation commands, and specify the **-qsmp** compiler option. Note that the compiler recognizes some of the directives if you specify either the **-qsmp** or **-qhot** compiler option.
- The XLF directives used to specify threadsafe code. To ensure that this functionality will be recognized by the compiler, compile your code using either the **xlf_r**, **xlf_r7**, **xlf90_r**, **xlf90_r7**, **xlf95_r** or **xlf95_r7** invocation commands.
- All comment form and noncomment form XLF directives from XL Fortran Version 7.1.
- Detailed descriptions of all data scope attribute clauses.

SMP and Thread-Safing Directives

The XLF Version 7.1 compiler provides a compiler option, **-qsmp=auto**, which instructs the compiler to automatically parallelize Fortran **DO** loops. This includes both **DO** loops coded explicitly by the user and **DO** loops generated by the compiler for array language constructs (for example, loops generated by **WHERE**, **FORALL**, and for array assignment). However, the compiler will only automatically parallelize loops that are *independent*; that is, loops whose iterations can be computed independently of any other iteration.

While automatic parallelization will be sufficient for some users, the SMP and thread-safing directives give you the option of providing additional information about the source code to the compiler. The compiler will use information given to it either during automatic parallelization, or to determine that certain parts of the program should be parallelized. For example, the **PARALLEL DO** directive specifies that the **DO** loop that is immediately following it should be run in parallel.

In XL Fortran these are the SMP and thread-safing directives that you can use:

- *Assertive*. These directives provide information to the compiler about the source code that the compiler would not necessarily be able to determine on its own.
  - **ASSERT** – See “ASSERT” on page 445

© Copyright IBM Corp. 1990, 2000

- **CNCALL** – See “CNCALL” on page 452
- **INDEPENDENT** – See “INDEPENDENT” on page 466
- **PERMUTATION** – See “PERMUTATION” on page 487

- *Prescriptive*. These directives instruct the compiler that it should apply some transformation to your code.
  - **ATOMIC** – See “ATOMIC” on page 448
  - **BARRIER** – See “BARRIER” on page 450
  - **CRITICAL** – See “CRITICAL / END CRITICAL” on page 453
  - **DO** (work-sharing) – See “DO / END DO” on page 455
  - **DO SERIAL** – See “DO SERIAL” on page 460
  - **FLUSH** – See “FLUSH” on page 462
  - **MASTER** – See “MASTER / END MASTER” on page 472
  - **ORDERED** – See “ORDERED / END ORDERED” on page 473
  - **PARALLEL** – See “PARALLEL / END PARALLEL” on page 476
  - **PARALLEL DO** – See “PARALLEL DO / END PARALLEL DO” on page 479
  - **PARALLEL SECTIONS** – See “PARALLEL SECTIONS / END PARALLEL SECTIONS” on page 483
  - **PREFETCH_BY_LOAD**, **PREFETCH_FOR_LOAD**, and **PREFETCH_FOR_STORE** – See “PREFETCH_BY_LOAD / PREFETCH_FOR_LOAD / PREFETCH_FOR_STORE” on page 488
  - **SCHEDULE** – See “SCHEDULE” on page 491
  - **SECTIONS** – See “SECTIONS / END SECTIONS” on page 498
  - **SINGLE** — See “SINGLE / END SINGLE” on page 502
  - **UNROLL** – See “UNROLL” on page 514
- *Thread-safing*. These directives allocate thread-specific **COMMON** areas at run time.
  - **THREADLOCAL** – See “THREADLOCAL” on page 507
  - **THREADPRIVATE** – See “THREADPRIVATE” on page 510

Noncomment and Comment Form Directives

All XL Fortran directives belong to one of two groups: noncomment form directives and comment form directives.

Noncomment Form Directives

Format

```
►►—directive——————————————————————————————————————————————————————►◄
```

directive
is one of the following directives:
@PROCESS – See “@PROCESS” on page 490
#line – See “#line” on page 470

EJECT – See “EJECT” on page 461
INCLUDE – See “INCLUDE” on page 464

Rules

The compiler always recognizes noncomment form directives. They cannot be continued.

Additional statements cannot be included on the same line as a directive.

Source format rules concerning white space apply to directive lines.

Comment Form Directives

Format

```
►►—trigger_head—trigger_constant—directive————————————————————————►◄
```

trigger_head
: is one of **!**, *, **C**, or **c** for fixed source form and **!** for free source form.

trigger_constant
: is **IBM*** by default. If you specify the **-qsmp** compiler option, **IBM***, **IBMT**, **SMP$**, **$OMP**, and **IBMP** are recognized by default. You can define other *trigger_constants* with the **-qdirective** compiler option. See the *User's Guide* for more details.

directive
: is one of the following directives:
ASSERT – See “ASSERT” on page 445
ATOMIC – See “ATOMIC” on page 448
BARRIER – See “BARRIER” on page 450
CNCALL – See “CNCALL” on page 452
CRITICAL – See “CRITICAL / END CRITICAL” on page 453
DO (work-sharing) – See “DO / END DO” on page 455
DO SERIAL – See “DO SERIAL” on page 460
END CRITICAL – See “CRITICAL / END CRITICAL” on page 453
END DO – See “DO / END DO” on page 455
END MASTER – See “MASTER / END MASTER” on page 472
END ORDERED – See “ORDERED / END ORDERED” on page 473
END PARALLEL – See “PARALLEL / END PARALLEL” on page 476
END PARALLEL DO – See “PARALLEL DO / END PARALLEL DO” on page 479

END PARALLEL SECTIONS – See "PARALLEL SECTIONS / END PARALLEL SECTIONS" on page 483
END SECTIONS – See "SECTIONS / END SECTIONS" on page 498
END SINGLE – See "SINGLE / END SINGLE" on page 502
FLUSH – See "FLUSH" on page 462
INDEPENDENT – See "INDEPENDENT" on page 466
MASTER – See "MASTER / END MASTER" on page 472
ORDERED – See "ORDERED / END ORDERED" on page 473
PARALLEL – See "PARALLEL / END PARALLEL" on page 476
PARALLEL DO – See "PARALLEL DO / END PARALLEL DO" on page 479
PARALLEL SECTIONS – See "PARALLEL SECTIONS / END PARALLEL SECTIONS" on page 483
PERMUTATION – See "PERMUTATION" on page 487
PREFETCH_BY_LOAD, **PREFETCH_FOR_LOAD**, **PREFETCH_FOR_STORE–** See "PREFETCH_BY_LOAD / PREFETCH_FOR_LOAD / PREFETCH_FOR_STORE" on page 488
SCHEDULE – See "SCHEDULE" on page 491
SECTION – See "SECTIONS / END SECTIONS" on page 498 and "PARALLEL SECTIONS / END PARALLEL SECTIONS" on page 483
SECTIONS– See "SECTIONS / END SECTIONS" on page 498
SINGLE — See "SINGLE / END SINGLE" on page 502
SOURCEFORM – See "SOURCEFORM" on page 506
THREADLOCAL – See "THREADLOCAL" on page 507
THREADPRIVATE – See "THREADPRIVATE" on page 510
UNROLL – See "UNROLL" on page 514

Definitions:

lexical extent
The lexical extent of a directive construct includes all the code that appears directly within the directive construct.

dynamic extent
The dynamic extent of a directive includes the lexical extent of the directive and all subprograms that are called from within the lexical extent.

parallel region
A parallel region is a region of code that is to be executed concurrently by multiple threads. You define a parallel region

by enclosing it within a **PARALLEL / END PARALLEL** construct, a **PARALLEL DO / END PARALLEL DO** construct, or a **PARALLEL SECTIONS / END PARALLEL SECTIONS** construct.

synchronization construct
A synchronization construct is a directive construct that controls the synchronization of threads within the construct. The following directive constructs are synchronization constructs:

- **ATOMIC**
- **BARRIER**
- **CRITICAL / END CRITICAL**
- **FLUSH**
- **MASTER / END MASTER**
- **ORDERED / END ORDERED**

work-sharing construct
A work-sharing construct divides the execution of the code that is enclosed within the construct among the members of the team that encounter it. Work-sharing constructs must be enclosed within the dynamic extent of a parallel region. The **DO / END DO** construct, the **SECTIONS / END SECTIONS** construct, and the **SINGLE / END SINGLE** construct are all work-sharing constructs. The **PARALLEL DO / END PARALLEL DO** and **PARALLEL SECTIONS / END PARALLEL SECTIONS** constructs are special cases of the work-sharing construct that also specify that the construct is a new parallel region. Since they define the enclosed region as a parallel region, they do not have to be enclosed within the dynamic extent of another parallel region.

Rules

The default value for the *trigger_constant* is **IBM***.

The compiler will always recognize comment form directives that use **IBM*** as the *trigger_constant*.

By default, the option **-qdirective=IBM*:IBMT** will be on when compiling using the **xlf_r**, **xlf_r7**, **xlf90_r**, **xlf90_r7**, **xlf95_r** or **xlf95_r7** invocation commands. By default, if you use the **-qsmp** compiler option in conjunction with one of these invocation commands, the option **-qdirective=IBM*:SMP$:$OMP:IBMP:IBMT** will be on. If you specify the **-qsmp=omp** option this will be as if you set the option **-qdirective=$OMP** on

by default. You can specify an alternate or additional *trigger_constant* with the **-qdirective** compiler option. See the **-qdirective** compiler option in the *User's Guide* for more details.

The compiler treats all comment form directives, with the exception of the default *trigger_constant* as comments, unless you define the appropriate *trigger_constant* using the **-qdirective** compiler option. As a result, code containing these directives is portable to non-SMP environments.

XLF supports the OpenMP specification, as understood and interpreted by IBM. In particular, XLF provides support for the following directives:
- **ATOMIC**
- **BARRIER**
- **CRITICAL / END CRITICAL**
- **DO/ END DO**
- **FLUSH**
- **MASTER / END MASTER**
- **ORDERED / END ORDERED**
- **PARALLEL / END PARALLEL**
- **PARALLEL DO / END PARALLEL DO**
- **PARALLEL SECTIONS / SECTION / END PARALLEL SECTIONS**
- **SECTIONS / SECTION / END SECTIONS**
- **SINGLE / END SINGLE**
- **THREADPRIVATE**

To ensure the greatest portability of code, we recommend that you use these directives whenever possible. You should use them with the OpenMP *trigger_constant*, **$OMP**; but you should not use this *trigger_constant* with any other directive.

XLF also includes the *trigger_constant*s **IBMP** and **IBMT**. The compiler recognizes **IBMP** if you compile using the **-qsmp** compiler option. You should use **IBMP** with the **SCHEDULE** directive, and **IBM** extensions to **OpenMP** directives. The compiler recognizes **IBMT** if you compile using the **-qthreaded** compiler option. **IBMT** is the default for the **xlf_r**, **xlf_r7**, **xlf90_r**, **xlf90_r7**, **xlf95_r** or **xlf95_r7** invocation commands; we recommend its use with the **THREADLOCAL** directive.

XLF directives include directives that are common to other vendors. If you use these directives in your code, you can enable whichever *trigger_constant* that vendor has selected. Specifying the trigger constant by using the **-qdirective** compiler option will enable the *trigger_constant* the vendor has

selected. Refer to the **-qdirective** compiler option in the *User's Guide* for details on specifying alternative *trigger_constant*s.

You can specify a directive as a free source form or fixed source form comment, depending on the current source form.

The *trigger_head* follows the rules of comment lines either in Fortran 90 free source form or fixed source form. If the *trigger_head* is **!**, it does not have to be in column 1. There must be no blanks between the *trigger_head* and the *trigger_constant*.

You can specify the *directive_trigger* (defined as the *trigger_head* combined with the *trigger_constant*, **!IBM*** for example) and any directive keywords in uppercase, lowercase, or mixed case.

You can specify inline comments on directive lines.

```
!SMP$ INDEPENDENT, NEW(i)    !This is a comment
```

A directive cannot follow another statement or another directive on the same line.

All comment form directives can be continued. You cannot embed a directive within a continued statement, nor can you embed a statement within a continued directive.

You must specify the *directive_trigger* on all continuation lines. However, the *directive_trigger* on a continuation line need not be identical to the *directive_trigger* that is used in the continued line. For example:

```
!SMP$ INDEPENDENT                         &
!IBM*&  , REDUCTION (X)                   &
!SMP$&  , NEW (I)
```

The above is equivalent to:

```
!SMP$ INDEPENDENT, REDUCTION (X), NEW (I)
```

provided both **IBM*** and **SMP$** are active *trigger_constant*s.

For more information, see "Lines and Source Formats" on page 16.

Fixed Source Form Rules: If the *trigger_head* is one of **C**, **c**, or *****, it must be in column 1.

The maximum length of the *trigger_constant* in fixed source form is 4 for directives that are continued on one or more lines. This rule applies to the continued lines only, not to the initial line. Otherwise, the maximum length of the *trigger_constant* is 15. We recommend that initial line triggers have a

maximum length of 4. The maximum allowable length of 15 is permitted for the purposes of backwards compatibility.

If the *trigger_constant* has a length of 4 or less, the first line of a comment directive must have either white space or a zero in column 6. Otherwise, the character in column 6 is part of the *trigger_constant*.

The *directive_trigger* of a continuation line of a comment directive must appear in columns 1-5. Column 6 of a continuation line must have a character that is neither white space nor a zero.

For more information, see “Fixed Source Form” on page 17.

Free Source Form Rules: The maximum length of the *trigger_constant* is 15.

An ampersand (&) at the end of a line indicates that the directive will continue. When you continue a directive line, a *directive_trigger* must appear at the beginning of all continuation lines. If you are beginning a continuation line with an ampersand, the *directive_trigger* must precede the ampersand. For example:

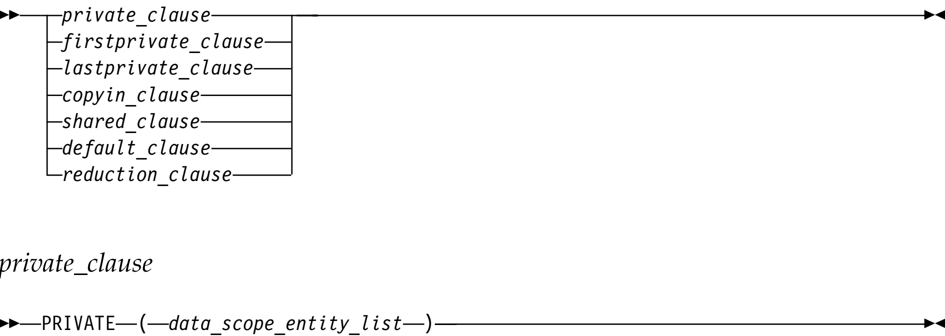
```
!IBM* INDEPENDENT                    &
!SMP$&  , REDUCTION (X)              &
!IBM*&  , NEW (I)
```

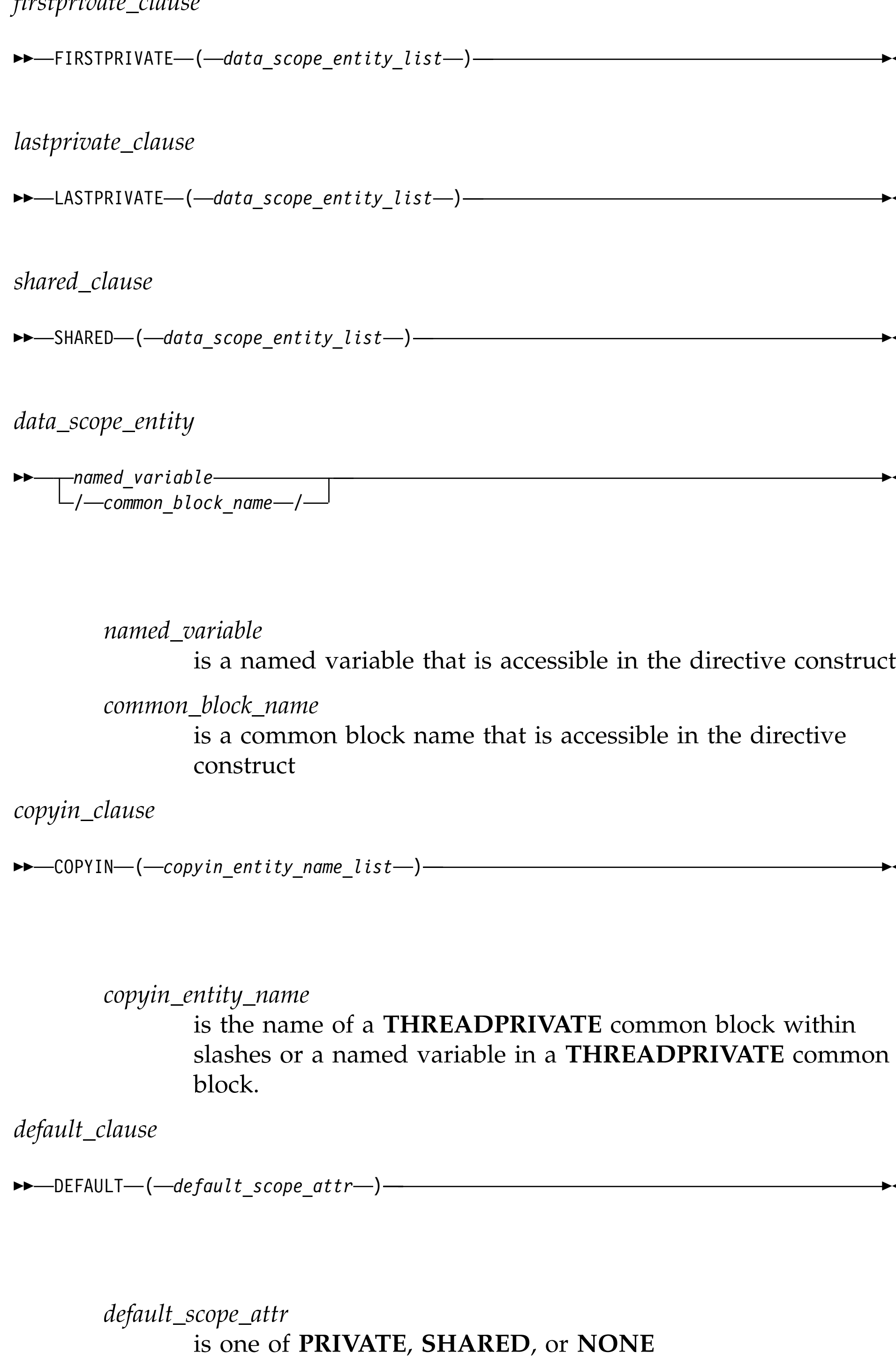
For more information, see “Free Source Form” on page 19.

Data Scope Attribute Clauses

Certain clauses allow you to specify the scope attributes of variables that you are using in parallel constructs. These clauses are called data scope attribute clauses.

Format





firstprivate_clause

```
>>--FIRSTPRIVATE--(--data_scope_entity_list--)-------------------><
```

lastprivate_clause

```
>>--LASTPRIVATE--(--data_scope_entity_list--)--------------------><
```

shared_clause

```
>>--SHARED--(--data_scope_entity_list--)-------------------------><
```

data_scope_entity

```
>>--+-named_variable-----------+---------------------------------><
    '-/--common_block_name--/--'
```

named_variable
: is a named variable that is accessible in the directive construct

common_block_name
: is a common block name that is accessible in the directive construct

copyin_clause

```
>>--COPYIN--(--copyin_entity_name_list--)------------------------><
```

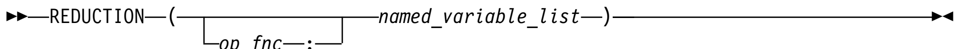
copyin_entity_name
: is the name of a **THREADPRIVATE** common block within slashes or a named variable in a **THREADPRIVATE** common block.

default_clause

```
>>--DEFAULT--(--default_scope_attr--)----------------------------><
```

default_scope_attr
: is one of **PRIVATE**, **SHARED**, or **NONE**

reduction_clause



op_fnc is a *reduction_op* or a *reduction_function* that appears in all **REDUCTION** statements involving this variable. Whether *op_fnc* is specified or not, only one **REDUCTION** operator or function must be used for that variable in the directive construct.

Rules

You cannot specify a variable or common block name more than once in a *data_scope_entity_list*. A variable or common block name must not appear in the *data_scope_entity_list* of more than one attribute clause on the same directive, with one exception. You can define a named variable or named common block as both **FIRSTPRIVATE** and **LASTPRIVATE** for the same directive construct.

If you specify a common block name on a data attribute clause, the common block name is expanded to its list of common block members by the compiler. Therefore, you cannot specify a common block name and a variable that is a member of that common block in the same *data_scope_entity_list*. Similarly, you cannot specify a common block name in the *data_scope_entity_list* of an attribute clause and a variable that is a member of that common block on another attribute clause of the same directive, except if the attribute clauses are the **FIRSTPRIVATE** and **LASTPRIVATE** clauses.

When individual members of a common block are privatized, the storage of the specified variable is no longer associated with the storage of the common block. Any variable that is storage associated with a member of a named common block that has the **FIRSTPRIVATE**, **PRIVATE**, or **LASTPRIVATE** attribute is undefined in the parallel construct.

You cannot specify a variable in a **PRIVATE**, **FIRSTPRIVATE** or **LASTPRIVATE** clause of a parallel construct if:

- the variable appears in a namelist statement, variable format expression or in an expression for a statement function definition, and,
- you reference the statement function, the variable format expression through formatted I/O, or the namelist through namelist I/O, within the parallel construct.

If you do not specify a data scope clause, the default scope for variables affected by the directive construct is **SHARED**.

Local variables without the **SAVE** or **STATIC** attributes in referenced subprograms in the dynamic extent of a parallel region have an implicit **PRIVATE** attribute. If you declare a local variable with the **SAVE** or **STATIC** attribute in a procedure that is called from a parallel region, the variable has an implicit **SHARED** attribute. Common blocks and modules in referenced subprograms in the dynamic extent of a parallel region have an implicit **SHARED** attribute, unless they are **THREADLOCAL** or **THREADPRIVATE** common blocks.

If there is a call to an MPI routine that does non-blocking communication in the directive construct, and any argument to the MPI routine is **FIRSTPRIVATE**, **LASTPRIVATE**, or **PRIVATE**, the MPI communication must complete before the end of the construct.

If one of the entities involved in an asynchronous I/O operation is a **FIRSTPRIVATE**, **LASTPRIVATE**, or **PRIVATE** variable; a subobject of a **FIRSTPRIVATE**, **LASTPRIVATE**, or **PRIVATE** variable; or a pointer that is associated with a **FIRSTPRIVATE**, **LASTPRIVATE**, or **PRIVATE** variable, the matching **WAIT** statement must be executed before the end of the thread.

A variable or a subobject of a variable in the *data_scope_entity_list* of a **PRIVATE**, **FIRSTPRIVATE** or **LASTPRIVATE** clause of a **PARALLEL**, **PARALLEL DO**, **DO**, **PARALLEL SECTIONS**, **SECTIONS** or **SINGLE** directive may have the **POINTER** attribute. Such a pointer has an undefined association status when a thread is created. The pointer also has an undefined association status when a thread is destroyed, unless the variable appears in the **LASTPRIVATE** clause. If the variable appears in the **LASTPRIVATE** clause, the pointer retains its association status at the end of the last iteration or **SECTION**.

Note: XL Fortran allows Fortran 90 pointers in **FIRSTPRIVATE** and **LASTPRIVATE** clauses; however, the OpenMP specification does not. Therefore, if you want your programs to be fully compliant with the OpenMP specification, you should not use Fortran 90 **POINTERS** or integer **POINTERS** in a **FIRSTPRIVATE** or **LASTPRIVATE** clause.

While a parallel or work-sharing directive construct is running, a variable or subobject of a variable that appears in a **PRIVATE**, **FIRSTPRIVATE**, **LASTPRIVATE** or **REDUCTION** clause of the directive must not be referenced, become defined, become undefined, have its association status or allocation status changed, or appear as an actual argument:
- In a scoping unit other than the one in which the directive construct appears; or
- In a variable format expression.

Variables that you specify as **REDUCTION** or **LASTPRIVATE** to a parallel construct become defined at the end of the construct. If you have concurrent definitions or uses of **REDUCTION** or **LASTPRIVATE** variables on multiple threads, you must ensure that the threads are synchnonized at the end of the construct when the variables become defined. For example, if multiple threads encounter a **PARALLEL** construct with a **REDUCTION** variable, you must synchronize the threads when they reach the **END PARALLEL** directive, because the **REDUCTION** variable becomes defined at **END PARALLEL**. Therefore the whole **PARALLEL** construct must be enclosed within a synchronization construct.

You cannot specify a **THREADPRIVATE** common block or the variables that comprise that block in a **PRIVATE**, **FIRSTPRIVATE**, **SHARED**, or **REDUCTION** clause.

You can declare a variable as **PRIVATE**, **FIRSTPRIVATE**, **LASTPRIVATE**, or **REDUCTION**, even if that variable is already storage associated with other variables. Storage association may exist for variables declared in **EQUIVALENCE** statements or in **COMMON** blocks. If a variable is storage associated with a **PRIVATE**, **FIRSTPRIVATE**, **LASTPRIVATE**, or **REDUCTION** variable, then:

- The contents, allocation status and association status of the variable that is storage associated with the **PRIVATE**, **FIRSTPRIVATE**, **LASTPRIVATE** or **REDUCTION** variable are undefined on entry to the parallel construct.
- The allocation status, association status and the contents of the associated variable become undefined if you define the **PRIVATE**, **FIRSTPRIVATE**, **LASTPRIVATE** or **REDUCTION** variable or if you define that variable's allocation or association status.
- The allocation status, association status and the contents of the **PRIVATE**, **FIRSTPRIVATE**, **LASTPRIVATE** or **REDUCTION** variable become undefined if you define the associated variable or if you define the associated variable's allocation or association status.

You cannot access the name of a common block by use association or host association. Thus, a named common block can only appear in a data attribute clause if the common block is declared in the scoping unit that contains the directive. However, you can access the variables in the common block by use association or host association. For more information, see "Host Association" on page 137 and "Use Association" on page 139.

Examples

Example 1: The following valid example demonstrates the proper use of a **PRIVATE** variable that is used to define a statement function. A commented

line shows the invalid use. Since J appears in a statement function, the statement function cannot be referenced within the parallel construct for which J is **PRIVATE**.

```
      INTEGER :: ARR(10), J = 17
      ISTFNC() = J

!$OMP PARALLEL DO PRIVATE(J)
      DO I = 1, 10
        ARR(I) = J
      ! ARR(I) = ISTFNC() **ERROR**   A reference to ISTFNC would
                                    ! make the PRIVATE(J) clause
                                    ! invalid.
      END DO
      PRINT *, ARR
      END
```

Example 2: In the following example, ABC and DEF are storage-associated, and ABC is a **LASTPRIVATE** variable. The example shows an instance of how ABC and DEF may become undefined.

```
      INTEGER:: ABC(10), DEF(10), INDX1
      EQUIVALENCE(ABC(9),DEF(1))

!$OMP PARALLEL DO PRIVATE(INDX1) LASTPRIVATE(ABC)
      DO INDX1= 1, 10
        ABC(INDX1) = INDX1 + 1      ! ABC IS REFERENCED, SO DEF
                                    ! BECOMES UNDEFINED.
        DEF(INDX1) = DEF(INDX1)-1   ! DEF'N OF DEF IS UNDEFINED.
      END DO

      PRINT *, ABC       ! LASTPRIVATE ABC IS UNDEFINED,
                         ! BECAUSE DEF WAS REFERENCED.
      PRINT *, DEF       ! DEF IS ALSO UNDEFINED SINCE
                         ! LASTPRIVATE ABC IS UNDEFINED.
```

Example 3: In the following valid example, the common block /ABC/ cannot be accessed by host or use association, so it cannot be specified on the **COPYIN** clause. However, the variable I, which is in the common block /ABC/, can be specified in the **COPYIN** clause.

```
      MODULE MOD
        COMMON /ABC/ I
!$OMP   THREADPRIVATE (/ABC/)
      END MODULE MOD

      PROGRAM T
        USE MOD
!$OMP   PARALLEL COPYIN(I)    ! COPYIN(/ABC/) WOULD BE INVALID HERE
! ...
        I=0
!$OMP   END PARALLEL
      END PROGRAM
```

List of Data Scope Attribute Clauses

The data scope attribute clauses are:

PRIVATE*(data_scope_entity_list)*

If you specify the **PRIVATE** clause on one of the directives listed below, each thread in a team has its own uninitialized local copy of the variables and common blocks in *data_scope_entity_list*.

You should specify a variable with the **PRIVATE** attribute if its value is calculated by a single thread and that value is not dependent on any other thread, if it is defined before it is used in the construct, and if its value is not used after the construct ends. Copies of the **PRIVATE** variable exist, locally, on each thread. Each thread receives its own uninitialized copy of the **PRIVATE** variable. A **PRIVATE** variable has an undefined value or association status on entry to, and exit from, the directive construct. All thread variables within the lexical extent of the directive construct have the **PRIVATE** attribute by default.

A variable in the **PRIVATE** clause must not be any of the following elements:
- A pointee
- An assumed-size array
- An assumed-shape array
- A **THREADLOCAL** common block variable
- A **THREADPRIVATE** common block variable

A variable name in the *data_scope_entity_list* of the **PRIVATE** clause can be an allocatable array. It must not be allocated on initial entry to the directive construct, and you must allocate and deallocate the array for every thread that executes the construct.

Local variables without the **SAVE** or **STATIC** attributes in referenced subprograms in the dynamic extent of a directive construct have an implicit **PRIVATE** attribute.

The **PRIVATE** clause applies to the following directives:
- **DO**
- **PARALLEL**
- **PARALLEL DO**
- **PARALLEL SECTIONS**
- **SECTIONS**
- **SINGLE**

FIRSTPRIVATE*(data_scope_entity_list)*

If you use the **FIRSTPRIVATE** clause, each thread has its own initialized local copy of the variables and common blocks in *data_scope_entity_list*.

The **FIRSTPRIVATE** clause can be specified for the same variables as the **PRIVATE** clause, and functions in a manner similar to the **PRIVATE** clause. The exception is the status of the variable upon entry into the directive construct; the **FIRSTPRIVATE** variable exists and is initialized for each thread entering the directive construct.

A variable in a **FIRSTPRIVATE** clause must not be any of the following elements:
- A pointee
- An assumed-size array
- An assumed-shape array
- A **THREADLOCAL** common block variable
- A **THREADPRIVATE** common block variable
- An allocatable array

The **FIRSTPRIVATE** clause applies to the following directives:
- **DO**
- **PARALLEL**
- **PARALLEL DO**
- **PARALLEL SECTIONS**
- **SECTIONS**
- **SINGLE**

LASTPRIVATE*(data_scope_entity_list)*

If you use the **LASTPRIVATE** clause, each variable and common block in *data_scope_entity_list* is **PRIVATE**, and the last value of each variable in *data_scope_entity_list* can be referred to outside of the construct of the directive. If you use the **LASTPRIVATE** clause with **DO** or **PARALLEL DO**, the last value is the value of the variable after the last sequential iteration of the loop. If you use the **LASTPRIVATE** clause with **SECTIONS** or **PARALLEL SECTIONS**, the last value is the value of the variable after the last **SECTION** of the construct. If the last iteration of the loop or last section of the construct does not define a **LASTPRIVATE** variable, the variable is undefined after the loop or construct.

The **LASTPRIVATE** clause functions in a manner similar to the **PRIVATE** clause and you should specify it for variables that match the same criteria. The exception is in the status of the variable on exit from the directive construct. The compiler determines the last value of the variable, and takes a copy of that value which it saves in the

named variable for use after the construct. A **LASTPRIVATE** variable is undefined on entry to the construct if it is not a **FIRSTPRIVATE** variable.

A variable in a **LASTPRIVATE** clause must not be any of the following elements:
- A pointee
- An assumed-size array
- An assumed-shape array
- A **THREADLOCAL** common block variable
- A **THREADPRIVATE** common block variable
- An allocatable array

If you specify a variable as **LASTPRIVATE** on a work-sharing directive, and you have specified a **NOWAIT** clause on that directive, you cannot use that variable prior to a barrier.

The **LASTPRIVATE** clause applies to the following directives:
- **DO**
- **PARALLEL DO**
- **PARALLEL SECTIONS**
- **SECTIONS**

Example:

The following example shows the proper use of a **LASTPRIVATE** variable after a **NOWAIT** clause.

```
!$OMP PARALLEL
!$OMP   DO LASTPRIVATE(K)

        DO I=1,10
            K=I+1
        END DO

!$OMP   END DO NOWAIT

! PRINT *, K **ERROR** ! The reference to K must occur after a
                       ! barrier.
!$OMP BARRIER
      PRINT *, K       ! This reference to K is legal.
!$OMP END PARALLEL
      END
```

REDUCTION*(op_fnc : named_variable_list)*

The **REDUCTION** clause asserts that updates to named variables will occur within **REDUCTION** statements in the directive construct. Furthermore, the intermediate values of the **REDUCTION** variables are not used within the parallel section, other than in the updates

themselves. Thus, the value of the **REDUCTION** variable after the construct is the result of a reduction tree.

Any variable you specify in a **REDUCTION** clause of a work-sharing construct must be shared in the enclosing **PARALLEL** construct.

If you specify *op_fnc* for the **REDUCTION** clause, each variable in the *named_variable_list* must be scalar and can only occur in a **REDUCTION** statement within the lexical extent of the directive construct. You must specify *op_fnc* if the directive uses the *trigger_constant* **$OMP**.

If you use a **REDUCTION** clause on a construct that has a **NOWAIT** clause, the **REDUCTION** variable remains undefined until a barrier synchronization has been performed to ensure that all threads have completed the **REDUCTION** clause.

A **REDUCTION** variable must not occur in a **FIRSTPRIVATE**, **PRIVATE** or **LASTPRIVATE** clause of another construct within the dynamic extent of the construct in which it appeared as a **REDUCTION** variable.

The canonical initialization value of each of the operators and intrinsics are shown in the following table. The actual initialization value will be consistent with the data type of your corresponding **REDUCTION** variable.

Operator/Intrinsic	Initialization
+	0
*	1
-	0
.AND.	.TRUE.
.OR.	.FALSE.
.EQV.	.TRUE.
.NEQV.	.FALSE.
.XOR.	.FALSE.
MAX	Smallest representable number
MIN	Largest representable number
IAND	All bits on
IOR	0
IEOR	0

A **REDUCTION** statement can have one of the following forms:

```
>>--reduction_var_ref--=--expr--reduction_op--reduction_var_ref--------------><
```

```
>>--reduction_var_ref--=--reduction_var_ref--reduction_op--expr--------------><
```

```
>>--reduction_var_ref =--reduction_function--(expr,--reduction_var_ref)------><
```

```
>>--reduction_var_ref =--reduction_function--(reduction_var_ref,--expr)------><
```

where:

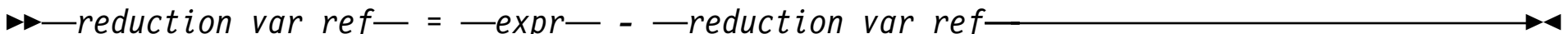
reduction_var_ref
: is a variable or subobject of a variable that appears in a **REDUCTION** clause

reduction_op
: is one of: **+**, **–**, *****, **.AND.**, **.OR.**, **.EQV.**, **.NEQV.**, or **.XOR.**

reduction_function
: is one of: **MAX**, **MIN**, **IAND**, **IOR**, or **IEOR**

The following rules apply to **REDUCTION** statements:

1. A variable in the **REDUCTION** clause must only occur in a **REDUCTION** statement within the directive construct on which the **REDUCTION** clause appears.
2. The two *reduction_var_ref*s that appear in a **REDUCTION** statement must be lexically identical.
3. You cannot use the following form of the **REDUCTION** statement:



The **REDUCTION** clause specifies named variables that appear in reduction operations. The compiler will maintain local copies of such variables, but will combine them upon exit from the construct. The intermediate values of the **REDUCTION** variables are combined in random order, dependent on which threads finish their calculations first. Therefore, there is no guarantee that bit-identical results will be obtained from one parallel run to another. This is true even if the parallel runs use the same number of threads, scheduling type, and chunk size.

A variable in the **REDUCTION** clause must be of intrinsic type. A variable in the **REDUCTION** clause, or any element thereof, must not be any of the following:

- A pointee
- An assumed-size array
- An assumed-shape array
- A **THREADLOCAL** common block variable
- A **THREADPRIVATE** common block variable
- An Allocatable array
- A Fortran 90 **POINTER**

These rules describe the use of **REDUCTION** on OpenMP directives. If you are using the **REDUCTION** clause on the **INDEPENDENT** directive, see "INDEPENDENT" on page 466.

The OpenMP implementation of the **REDUCTION** clause applies to:

- **DO**
- **PARALLEL**
- **PARALLEL DO**
- **PARALLEL SECTIONS**
- **SECTIONS**

COPYIN*(copyin_entity_name_list)*

If you specify the **COPYIN** clause, the master thread's copy of each common entity in the *copyin_entity_name_list* is duplicated at the beginning of the parallel region privately for each of the other threads in the team that will execute the parallel region.

If you specify a **COPYIN** clause, you cannot:

- specify the same common block name more than once in a *copyin_entity_name_list*.
- specify the same common block name in separate **COPYIN** clauses on the same directive.
- specify both a common block name and any variable within that same named common block in a *copyin_entity_name_list*.
- specify both a common block name and any variable within that same named common block in different **COPYIN** clauses on the same directive.

When the master thread of a team of threads reaches a directive containing the **COPYIN** clause, it is the compiler's responsibility to ensure that each thread's private copy of a variable or common block specified in the **COPYIN** clause has the same value as the master thread's copy.

The **COPYIN** clause applies to:

- **PARALLEL**
- **PARALLEL DO**
- **PARALLEL SECTIONS**

DEFAULT(*default_scope_attr)*

If you specify the **DEFAULT** clause, all variables in the lexical extent of the parallel construct will have a scope attribute of *default_scope_attr*.

If you specify **DEFAULT(NONE)**, there is no default scope attribute. Therefore, you must explicitly list each variable you use in the lexical extent of the parallel construct in a data scope attribute clause on the parallel construct, unless the variable is:

- **THREADPRIVATE**
- A pointee
- A loop iteration variable used only as a loop iteration variable for:
  - Sequential loops in the lexical extent of the parallel region, or,
  - Parallel do loops that bind to the parallel region
- A variable that is only used in work-sharing constructs that bind to the parallel region, and is specified in a data scope attribute clause for each of the work-sharing constructs.

The **DEFAULT** clause specifies that all variables in the parallel construct share the same default scope attribute of either **PRIVATE**, **SHARED**, or no default scope attribute.

If you specify **DEFAULT(NONE)** on a directive there will be no implicit **DEFAULT** scope attribute as to whether variables are **PRIVATE** or **SHARED**. In this situation you must specify all named variables and all the leftmost names of referenced array sections, array elements, structure components, or substrings in the lexical extent of the directive construct in a **PRIVATE**, **FIRSTPRIVATE**, **SHARED**, or **REDUCTION** clause.

If you specify **DEFAULT(PRIVATE)** on a directive, all named variables and all leftmost names of referenced array sections, array elements, structure components, or substrings in the lexical extent of the directive construct, including common block and use associated variables, but excluding **POINTEE**s and **THREADLOCAL** common blocks, have a **PRIVATE** attribute to a thread as if they were listed explicity in a **PRIVATE** clause.

If you specify **DEFAULT(SHARED)** on a directive, all named variables and all lefmost names of referenced array sections, array

elements, structure components, or substrings in the lexical extent of the directive construct, excluding **POINTEE**s have a **SHARED** attribute to a thread as if they were listed explicity in a **SHARED** clause.

The default behaviour will be **DEFAULT(SHARED)** if you do not explicitly indicate a **DEFAULT** clause on a directive.

The **DEFAULT** clause applies to:

- **PARALLEL**
- **PARALLEL DO**
- **PARALLEL SECTIONS**

Example:

The following example demonstrates the use of **DEFAULT(NONE)**, and some of the rules for specifying the data scope attributes of variables in the parallel region.

```
      PROGRAM MAIN
        COMMON /COMBLK/ ABC(10), DEF

          ! THE LOOP ITERATION VARIABLE, I, IS NOT
          ! REQUIRED TO BE IN DATA SCOPE ATTRIBUTE CLAUSE

!$OMP   PARALLEL DEFAULT(NONE) SHARED(ABC)

          ! DEF IS SPECIFIED ON THE WORK-SHARING DO AND IS NOT
          ! REQUIRED TO BE SPECIFIED IN A DATA SCOPE ATTRIBUTE
          ! CLAUSE ON THE PARALLEL REGION.

!$OMP   DO FIRSTPRIVATE(DEF)
        DO I=1,10
          ABC(I) = DEF
        END DO
!$OMP   END PARALLEL
      END
```

SHARED_(data_scope_entity_list)_

All sections use the same copy of the variables and common blocks you specify in _data_scope_entity_list_.

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.

A variable in the **SHARED** clause must not be either:

- A pointee
- A **THREADLOCAL** common block variable
- A **THREADPRIVATE** common block variable.

If a **SHARED** variable, a subobject of a **SHARED** variable, or an object associated with a **SHARED** variable or subobject of a **SHARED** variable appears as an actual argument in a reference to a non-intrinsic procedure and:

- The actual argument is an array section with a vector subscript; or
- The actual argument is
  - An array section,
  - An assumed-shape array, or,
  - A pointer array

  and the associated dummy argument is an explicit-shape or assumed-size array;

then any references to or definitions of the shared storage that is associated with the dummy argument by any other thread must be synchronized with the procedure reference. You can do this, for example, by placing the procedure reference after a **BARRIER**.

The **SHARED** clause applies to:

- **PARALLEL**
- **PARALLEL DO**
- **PARALLEL SECTIONS**

Example:

In the following example, the procedure reference with an array section actual argument is required to be synchronized with references to the dummy argument by placing the procedure reference in a critical section, because the associated dummy argument is an explicit-shape array.

```
      INTEGER:: ABC(10)
      I=2; J=5
!$OMP PARALLEL DEFAULT(NONE), SHARED(ABC,I,J)
!$OMP   CRITICAL
          CALL SUB1(ABC(I:J))    ! ACTUAL ARGUMENT IS AN ARRAY
                                 ! SECTION; THE PROCEDURE
                 ! REFERENCE MUST BE IN A CRITICAL SECTION.

!$OMP   END CRITICAL
!$OMP END PARALLEL
      CONTAINS
        SUBROUTINE SUB1(ARR)
          INTEGER:: ARR(1:4)
          DO I=1, 4
            ARR(I) = I
          END DO
        END SUBROUTINE
      END
```

Detailed Descriptions of Compiler Directives

The following is an alphabetical list of all compiler directives supported by XL Fortran. This section describes the following directives:

- "ASSERT"
- "ATOMIC" on page 448
- "BARRIER" on page 450
- "CNCALL" on page 452
- "CRITICAL / END CRITICAL" on page 453
- "DO / END DO" on page 455
- "DO SERIAL" on page 460
- "EJECT" on page 461
- "FLUSH" on page 462
- "INCLUDE" on page 464
- "INDEPENDENT" on page 466
- "#line" on page 470
- "MASTER / END MASTER" on page 472
- "ORDERED / END ORDERED" on page 473
- "PARALLEL DO / END PARALLEL DO" on page 479
- "PARALLEL / END PARALLEL" on page 476
- "PARALLEL SECTIONS / END PARALLEL SECTIONS" on page 483
- "PERMUTATION" on page 487
- "PREFETCH_BY_LOAD / PREFETCH_FOR_LOAD / PREFETCH_FOR_STORE" on page 488
- "@PROCESS" on page 490
- "SCHEDULE" on page 491
- "SECTIONS / END SECTIONS" on page 498
- "SINGLE / END SINGLE" on page 502
- "SOURCEFORM" on page 506
- "THREADLOCAL" on page 507
- "THREADPRIVATE" on page 510
- "UNROLL" on page 514

ASSERT

Purpose

The **ASSERT** directive provides information to the compiler about the characteristics of **DO** loops. This assists the compiler in optimizing the source code.

The directive only takes effect if you specify either the **-qsmp** or **-qhot** compiler option.

Format

```
►►—ASSERT—(—assertion_list—)——————————————————————————►◄
```

assertion
: is **ITERCNT***(n)* or **NODEPS**. **ITERCNT***(n)* and **NODEPS** are not mutually exclusive, and you can specify both for the same **DO** loop. You can use at most one of each argument for the same **DO** loop.

  ITERCNT*(n)*
  : where *n* specifies the number of iterations for a given **DO** loop. *n* must be a positive, scalar, integer initialization expression.

  NODEPS
  : specifies that no loop-carried dependencies exist within a given **DO** loop.

Rules

The first noncomment line (not including other directives) following the **ASSERT** directive must be a **DO** loop. This line cannot be an infinite **DO** or **DO WHILE** loop. The **ASSERT** directive applies only to the **DO** loop immediately following the directive, and not to any nested **DO** loops.

ITERCNT provides an estimate to the compiler about roughly how many iterations the **DO** loop will typically run. There is no requirement that the value be accurate; **ITERCNT** will only affect performance, never correctness.

When **NODEPS** is specified, the user is explicitly declaring to the compiler that no loop-carried dependencies exist within the **DO** loop or any procedures invoked from within the **DO** loop. A loop-carried dependency involves two iterations within a **DO** loop interfering with one another. Interference occurs in the following situations:

- Two operations that define, undefine, or redefine the same atomic object (data that has no subobjects) interfere.
- Definition, undefinition, or redefinition of an atomic object interferes with any use of the value of the object.
- Any operation that causes the association status of a pointer to become defined or undefined interferes with any reference to the pointer or any other operation that causes the association status to become defined or undefined.
- Transfer of control outside the **DO** loop or execution of an **EXIT**, **STOP**, or **PAUSE** statement interferes with all other iterations.
- Any two input/output (I/O) operations associated with the same file or external unit interfere with each other. The exceptions to this rule are:

  – If the two I/O operations are two **INQUIRE** statements; or
  – If the two I/O operations are two distinct records of a direct access file.
- A change in the allocation status of an allocatable object between iterations causes interference.

It is possible for two complementary **ASSERT** directives to apply to any given **DO** loop. However, an **ASSERT** directive cannot be followed by a contradicting **ASSERT** directive for a given **DO** loop:

```
!SMP$ ASSERT (ITERCNT(10))
!SMP$ INDEPENDENT, REDUCTION (A)
!SMP$ ASSERT (ITERCNT(20))     ! invalid
      DO I = 1, N
          A(I) = A(I) * I
      END DO
```

In the example above, the **ASSERT(ITERCNT(20))** directive contradicts the **ASSERT(ITERCNT(10))** directive and is invalid.

The **ASSERT** directive overrides the **-qassert** compiler option for the **DO** loop on which the **ASSERT** directive is specified.

Examples

Example 1:

```
! An example of the ASSERT directive with NODEPS.
         PROGRAM EX1
           INTEGER A(100)
  !SMP$    ASSERT (NODEPS)
           DO I = 1, 100
             A(I) = A(I) * FNC1(I)
           END DO
         END PROGRAM EX1

         FUNCTION FNC1(I)
           FNC1 = I * I
         END FUNCTION FNC1
```

Example 2:

```
! An example of the ASSERT directive with NODEPS and ITERCNT.
         SUBROUTINE SUB2 (N)
           INTEGER A(N)
    !SMP$  ASSERT (NODEPS,ITERCNT(100))
           DO I = 1, N
             A(I) = A(I) * FNC2(I)
           END DO
         END SUBROUTINE SUB2

         FUNCTION FNC2 (I)
           FNC2 = I * I
         END FUNCTION FNC2
```

Related Information

- ″-qassert Option″ in the *User's Guide*
- ″-qdirective Option″ in the *User's Guide*
- ″-qhot Option″ in the *User's Guide*
- ″-qsmp Option″ in the *User's Guide*
- "DO" on page 284
- "Loop Parallelization" on page 313

ATOMIC

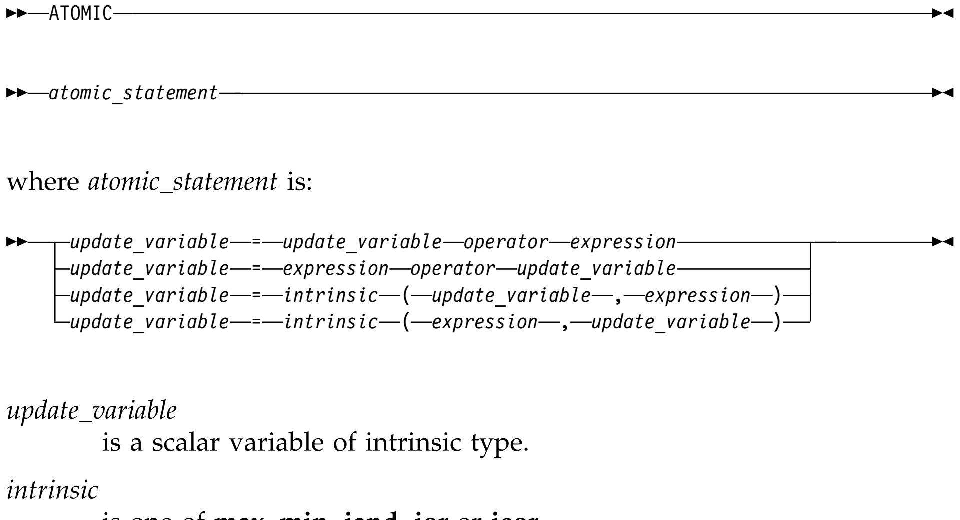
Purpose

You can use the **ATOMIC** directive to update a specific memory location safely within a parallel region. When you use **ATOMIC**, you ensure that only one thread is writing to the memory location at a time, avoiding errors which might occur from simultaneous writes to the same memory location.

Normally, you would protect a shared variable within a **CRITICAL** construct if it is being updated by more than one thread at a time. However, certain platforms support atomic operations for updating variables. For example, some platforms might support a hardware instruction that reads from a memory location, calculates something and writes back to the location all in one atomic action. The **ATOMIC** directive instructs the compiler to use an atomic operation whenever possible. Otherwise, the compiler will use some other mechanism to perform an atomic update.

The **ATOMIC** directive only takes effect if you specify the **-qsmp** compiler option.

Format



where *atomic_statement* is:

update_variable
: is a scalar variable of intrinsic type.

intrinsic
: is one of **max**, **min**, **iand**, **ior** or **ieor**.

operator
: is one of **+**, **–**, *****, **/**, **.AND.**, **.OR.**, **.EQV.**, **.NEQV.** or **.XOR.**

expression
: is a scalar expression that does not reference *update_variable*.

Rules

The **ATOMIC** directive applies only to the statement which immediately follows it.

The *expression* in an *atomic_statement* is not evaluated atomically. You must ensure that no race conditions exist in the calculation.

All references to the same *update_variable* that follow an **ATOMIC** directive must have the same data type and type parameters.

The function *intrinsic*, the operator *operator*, and the assignment must be the intrinsic function, operator and assignment and not a redefined intrinsic function, defined operator or defined assignment.

Examples

Example 1: In the following example, multiple threads are updating a counter. **ATOMIC** is used to ensure that no updates are lost.

```
      PROGRAM P
        R = 0.0
!$OMP   PARALLEL DO SHARED(R)
        DO I=1, 10
!$OMP     ATOMIC
          R = R + 1.0
        END DO
        PRINT *,R
      END PROGRAM P
```

Expected output:

```
10.0
```

Example 2: In the following example, an **ATOMIC** directive is required, because it is uncertain which element of array *Y* will be updated in each iteration.

```
      PROGRAM P
        INTEGER, DIMENSION(10) :: Y, INDEX
        INTEGER B
        Y = 5
        READ(*,*) INDEX, B
!$OMP   PARALLEL DO SHARED(Y)
        DO I = 1, 10
!$OMP     ATOMIC
```

```
    Y(INDEX(I)) = MIN(Y(INDEX(I)),B)
  END DO
  PRINT *, Y
END PROGRAM P
```

Input data:

```
10 10 8 8 6 6 4 4 2 2   4
```

Expected output:

```
5 4 5 4 5 4 5 4 5 4
```

Example 3: The following example is invalid, because you cannot use an **ATOMIC** operation to reference an array.

```
      PROGRAM P
           REAL ARRAY(10)
           ARRAY = 0.0
     !$OMP PARALLEL DO SHARED(ARRAY)
           DO I = 1, 10
     !$OMP   ATOMIC
             ARRAY = ARRAY + 1.0
           END DO
           PRINT *, ARRAY
           END PROGRAM P
```

Example 4: In the following invalid example, the **ATOMIC** operation attempts to reference *R* twice, which is illegal. The *expression* and *update_variable* cannot be identical.

```
      PROGRAM P
           R = 0.0
     !$OMP PARALLEL DO SHARED(R)
           DO I = 1, 10
     !$OMP   ATOMIC
             R = R + R
           END DO
           PRINT *, R
           END PROGRAM P
```

Related Information

- “CRITICAL / END CRITICAL” on page 453
- “PARALLEL DO / END PARALLEL DO” on page 479
- “PARALLEL / END PARALLEL” on page 476
- ″-qsmp Option″ in the *User's Guide*

BARRIER

Purpose

The **BARRIER** directive enables you to synchronize all threads in a team. When a thread encounters a **BARRIER** directive, it will wait until all other threads in the team reach the same point.

The **BARRIER** directive only takes effect if you specify the **-qsmp** compiler option.

Format

```
▸▸—BARRIER————————————————————————————————————————————————————▸◂
```

Rules

A **BARRIER** directive binds to the closest dynamically enclosing **PARALLEL** directive, if one exists.

A **BARRIER** directive cannot appear within the dynamic extent of the **CRITICAL**, **DO** (work-sharing), **MASTER**, **PARALLEL DO**, **PARALLEL SECTIONS**, **SECTIONS**, and **SINGLE** directives.

All threads in the team must encounter the **BARRIER** directive if any thread encounters it.

All **BARRIER** directives and work-sharing constructs must be encountered in the same order by all threads in the team.

In addition to synchronizing the threads in a team, the **BARRIER** directive implies the **FLUSH** directive. This means it causes thread visible variables to be written back to memory. Thread visible variables are variables that are or might be **SHARED** for the **PARALLEL** construct containing the **BARRIER** directive, and that are accessible in the scoping unit containing the **BARRIER** directive.

Examples

Example 1: An example of the **BARRIER** directive binding to the **PARALLEL** directive. Note: To calculate ″c″, we need to ensure that ″a″ and ″b″ have been completely assigned to, so threads need to wait.

```
      SUBROUTINE SUB1
      INTEGER A(1000), B(1000), C(1000)
!$OMP PARALLEL
!$OMP DO
      DO I = 1, 1000
        A(I) = SIN(I*2.5)
      END DO
!$OMP END DO NOWAIT
!$OMP DO
      DO J = 1, 10000
        B(J) = X + COS(J*5.5)
      END DO
!$OMP END DO NOWAIT

   ⋮
```

```
!$OMP BARRIER
      C = A + B
!$OMP END PARALLEL
      END
```

Example 2: An example of a **BARRIER** directive that incorrectly appears inside a **CRITICAL** section. This can result in a deadlock since only one thread can enter a **CRITICAL** section at a time.

```
!$OMP PARALLEL DEFAULT(SHARED)

!$OMP CRITICAL
      DO I = 1, 10
        X= X + 1
!$OMP   BARRIER
        Y= Y + I*I
      END DO
!$OMP END CRITICAL

!$OMP END PARALLEL
```

Related Information

- “FLUSH” on page 462
- “Loop Parallelization” on page 313
- ″-qdirective Option″ in the *User's Guide*
- ″-qsmp Option″ in the *User's Guide*
- “PARALLEL DO / END PARALLEL DO” on page 479
- “PARALLEL SECTIONS / END PARALLEL SECTIONS” on page 483
- “PARALLEL / END PARALLEL” on page 476

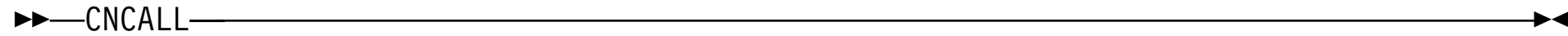
CNCALL

Purpose

When the **CNCALL** directive is placed before a **DO** loop, you are explicitly declaring to the compiler that no loop-carried dependencies exist within any procedure called from the **DO** loop.

This directive only takes effect if you specify either the **-qsmp** or **-qhot** compiler option.

Format



Rules

The first noncomment line (not including other directives) that is following the **CNCALL** directive must be a **DO** loop. This line cannot be an infinite **DO**

or **DO WHILE** loop. The **CNCALL** directive applies only to the **DO** loop that is immediately following the directive and not to any nested **DO** loops.

When specifying the **CNCALL** directive, you are explicitly declaring to the compiler that no procedures invoked within the **DO** loop have any loop-carried dependencies. If the **DO** loop invokes a procedure, separate iterations of the loop must be able to concurrently call that procedure. The **CNCALL** directive does not assert that other operations in the loop do not have dependencies, it is only an assertion about procedure references.

A loop-carried dependency occurs when two iterations within a **DO** loop interfere with one another. See "ASSERT" on page 445 for the definition of interference.

Examples

```
! An example of CNCALL where the procedure invoked has
! no loop-carried dependency but the code within the
! DO loop itself has a loop-carried dependency.
      PROGRAM EX3
        INTEGER A(100)
  !SMP$  CNCALL
        DO I = 1, N
          A(I) = A(I) * FNC3(I)
          A(I) = A(I) + A(I-1)    ! This has loop-carried dependency
        END DO
      END PROGRAM EX3

      FUNCTION FNC3 (I)
        FNC3 = I * I
      END FUNCTION FNC3
```

Related Information

- "INDEPENDENT" on page 466
- "-qdirective Option" in the *User's Guide*
- "-qhot Option" in the *User's Guide*
- "-qsmp Option" in the *User's Guide*
- "DO" on page 284
- "Loop Parallelization" on page 313

CRITICAL / END CRITICAL

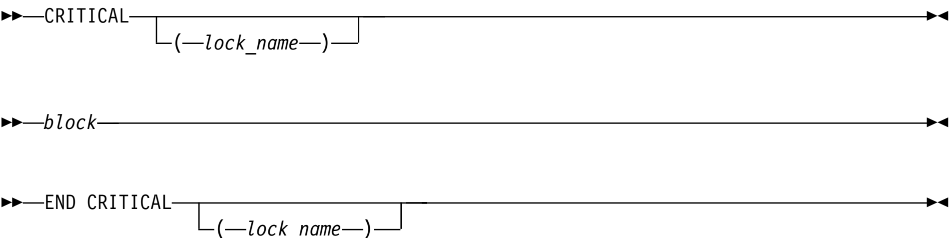
Purpose

The **CRITICAL** construct allows you to define independent blocks of code that are to be run by at most one thread at a time. It includes a **CRITICAL** directive that is followed by a block of code and ends with an **END CRITICAL** directive.

The **CRITICAL** and **END CRITICAL** directives only take effect if you specify the **-qsmp** compiler option.

CRITICAL / END CRITICAL

Format



lock_name
: provides a way of distinguishing different **CRITICAL** constructs of code.

block
: represents the block of code to be executed by at most one thread at a time.

Rules

The optional *lock_name* is a name with global scope. You must not use the *lock_name* to identify any other global entity in the same executable program.

If you specify the *lock_name* on the **CRITICAL** directive, you must specify the same *lock_name* on the corresponding **END CRITICAL** directive.

If you specify the same *lock_name* for more than one **CRITICAL** construct, the compiler will allow only one thread to execute any one of these **CRITICAL** constructs at any one time. **CRITICAL** constructs that have different *lock_names* may be run in parallel.

The same lock protects all **CRITICAL** constructs that do not have an explicit *lock_name*. In other words, the compiler will assign the same *lock_name*, thereby ensuring that only one thread enters any unnamed **CRITICAL** construct at a time.

The *lock_name* must not share the same name as any local entity of Class 1. For the definition of a local entity of Class 1, see "Local Entity" on page 134.

It is illegal to branch into or out of a **CRITICAL** construct.

The **CRITICAL** construct may appear anywhere in a program.

Although it is possible to nest a **CRITICAL** construct within a **CRITICAL** construct, a deadlock situation may result. The **–qsmp=rec_locks** compiler option can be used to prevent deadlocks. See *User's Guide* for more information.

The **CRITICAL** and **END CRITICAL** directives imply the **FLUSH** directive.

Examples

Example 1: Note that in this example the **CRITICAL** construct appears within a **DO** loop that has been marked with the **PARALLEL DO** directive.

```
      EXPR=0
!SMP$ PARALLEL DO PRIVATE (I)
      DO I = 1, 100
!SMP$   CRITICAL
          EXPR = EXPR + A(I) * I
!SMP$   END CRITICAL
      END DO
```

Example 2: An example specifying a *lock_name* on the **CRITICAL** construct.

```
!SMP$ PARALLEL DO PRIVATE(T)
      DO I = 1, 100
        T = B(I) * B(I-1)
!SMP$   CRITICAL (LOCK)
          SUM = SUM + T
!SMP$   END CRITICAL (LOCK)
      END DO
```

Related Information

- “FLUSH” on page 462
- “Loop Parallelization” on page 313
- “PARALLEL DO / END PARALLEL DO” on page 479
- “PARALLEL SECTIONS / END PARALLEL SECTIONS” on page 483
- “PARALLEL / END PARALLEL” on page 476
- ″-qdirective Option″ in the *User's Guide*
- ″-qsmp Option″ in the *User's Guide*
- “Local Entity” on page 134

DO / END DO

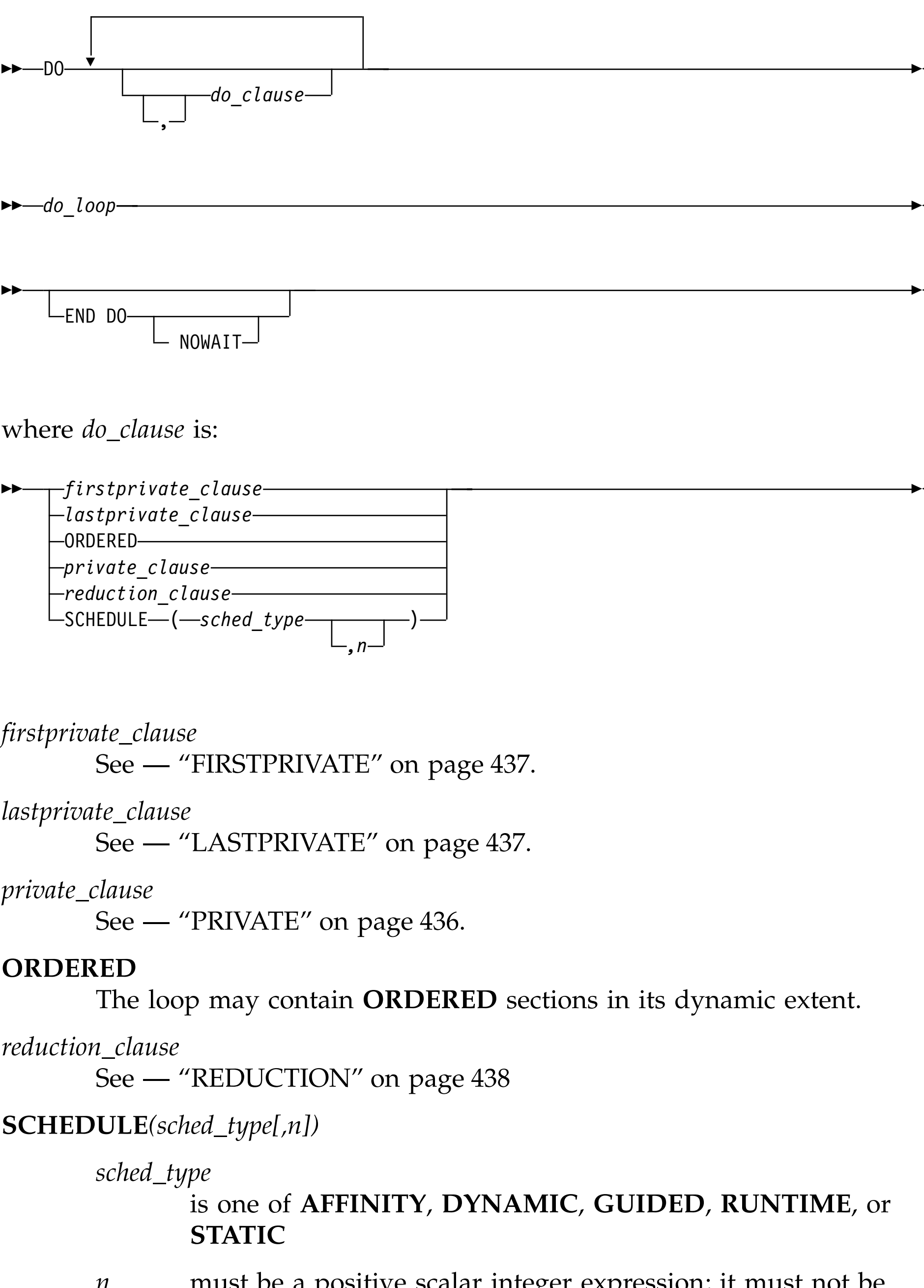
Purpose

The **DO** (work-sharing) construct enables you to divide the execution of the loop among the members of the team that encounter it. The **END DO** directive enables you to indicate the end of a **DO** loop that is specified by the **DO** (work-sharing) directive.

The **DO** (work-sharing) and **END DO** directives only take effect when you specify the **-qsmp** compiler option.

DO / END DO

Format



firstprivate_clause
: See — “FIRSTPRIVATE” on page 437.

lastprivate_clause
: See — “LASTPRIVATE” on page 437.

private_clause
: See — “PRIVATE” on page 436.

ORDERED
: The loop may contain **ORDERED** sections in its dynamic extent.

reduction_clause
: See — “REDUCTION” on page 438

SCHEDULE(*sched_type[,n]*)

sched_type
: is one of **AFFINITY**, **DYNAMIC**, **GUIDED**, **RUNTIME**, or **STATIC**

n
: must be a positive scalar integer expression; it must not be specified for the **RUNTIME** *sched_type*. See “SCHEDULE” on page 491

page 491 for definitions of these scheduling types. If you are using the *trigger_constant* **$OMP**, do not specify the scheduling type **AFFINITY**.

Rules

The first noncomment line (not including other directives) that follows the **DO** (work-sharing) directive must be a **DO** loop. This line cannot be an infinite **DO** or **DO WHILE** loop. The **DO** (work-sharing) directive applies only to the **DO** loop that is immediately following the directive, and not to any nested **DO** loops.

The **END DO** directive is optional. If you use the **END DO** directive, it must immediately follow the end of the **DO** loop.

You may have a **DO** construct that contains several **DO** statements. If the **DO** statements share the same **DO** termination statement, and an **END DO** directive follows the construct, you can only specify a work-sharing **DO** directive for the outermost **DO** statement of the construct.

If you specify **NOWAIT** on the **END DO** directive, a thread that completes its iterations of the loop early will proceed to the instructions following the loop. The thread will not wait for the other threads of the team to complete the **DO** loop. If you do not specify **NOWAIT** on the **END DO** directive, each thread will wait for all other threads within the same team at the end of the **DO** loop.

If you do not specify the **NOWAIT** clause, the **END DO** directive implies the **FLUSH** directive.

All threads in the team must encounter the **DO** (work-sharing) directive if any thread encounters it. A **DO** loop must have the same loop boundary and step value for each thread in the team. All work-sharing constructs and **BARRIER** directives that are encountered must be encountered in the same order by all threads in the team.

A **DO** (work-sharing) directive must not appear within the dynamic extent of a **CRITICAL** or **MASTER** construct. In addition, it must not appear within the dynamic extent of a **PARALLEL SECTIONS** construct, work-sharing construct, or **PARALLEL DO** loop, unless it is within the dynamic extent of a **PARALLEL** construct.

You cannot follow a **DO** (work-sharing) directive by another **DO** (work-sharing) directive. You can only specify one **DO** (work-sharing) directive for a given **DO** loop.

The **DO** (work-sharing) directive cannot appear with either an **INDEPENDENT** or **DO SERIAL** directive for a given **DO** loop. The **SCHEDULE** clause may appear at most once in a **DO** (work-sharing) directive.

Examples

Example 1: An example of several independent **DO** loops within a **PARALLEL** construct. No synchronization is performed after the first work-sharing **DO** loop, because **NOWAIT** is specified on the **END DO** directive.

```
!$OMP PARALLEL
!$OMP DO
      DO I = 2, N
        B(I)= (A(I) + A(I-1)) / 2.0
      END DO
!$OMP END DO NOWAIT
!$OMP DO
      DO J = 2, N
         C(J) = SQRT(REAL(J*J))
      END DO
!$OMP END DO
      C(5) = C(5) + 10
!$OMP END PARALLEL
      END
```

Example 2: An example of **SHARED**, and **SCHEDULE** clauses.

```
!$OMP PARALLEL SHARED(A)
!$OMP DO SCHEDULE(STATIC,10)
      DO I = 1, 1000
         A(I) = 1 * 4
      END DO
!$OMP END DO
!$OMP END PARALLEL
```

Example 3: An example of both a **MASTER** and a **DO** (work-sharing) directive that bind to the closest enclosing **PARALLEL** directive.

```
!$OMP PARALLEL DEFAULT(PRIVATE)
      Y = 100
!$OMP MASTER
      PRINT *, Y
!$OMP END MASTER
!$OMP DO
      DO I = 1, 10
         X(I) = I
         X(I) = X(I) + Y
      END DO
!$OMP END PARALLEL
      END
```

Example 4: An example of both the **FIRSTPRIVATE** and the **LASTPRIVATE** clauses on **DO** (work-sharing) directives.

```
      X = 100

!$OMP PARALLEL PRIVATE(I), SHARED(X,Y)
!$OMP DO FIRSTPRIVATE(X), LASTPRIVATE(X)
      DO I = 1, 80
         Y(I) = X + I
         X = I
      END DO
!$OMP END PARALLEL
      END
```

Example 5: A valid example of the **END DO** directive.

```
         REAL A(100), B(2:100), C(100)
!$OMP PARALLEL
!$OMP DO
         DO I = 2, 100
             B(I) = (A(I) + A(I-1))/2.0
         END DO
!$OMP END DO NOWAIT
!$OMP DO
         DO 200 J = 1, 100
            C(J) = SQRT(REAL(J*J))
200      CONTINUE
!$OMP END DO
!$OMP END PARALLEL
      END
```

Example 6: A valid example of a work-sharing **DO** directive applied to nested **DO** statements with a common **DO** termination statement.

```
!$OMP DO               ! A work-sharing DO directive can only
                       ! precede the outermost DO statement.
      DO 100 I= 1,10

! !$OMP DO **Error**   ! Placing the OMP DO directive here is
                       ! invalid

         DO 100 J= 1,10

!          ...

100   CONTINUE
!$OMP END DO
```

Related Information

- “DO” on page 284
- “FLUSH” on page 462
- “INDEPENDENT” on page 466
- “Loop Parallelization” on page 313

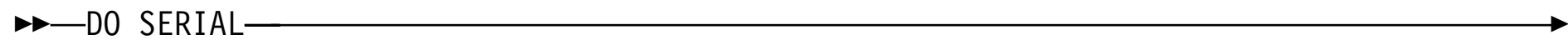
- “ORDERED / END ORDERED” on page 473
- “PARALLEL / END PARALLEL” on page 476
- “PARALLEL DO / END PARALLEL DO” on page 479
- “PARALLEL SECTIONS / END PARALLEL SECTIONS” on page 483
- “SCHEDULE” on page 491
- ″-qdirective Option″ in the *User's Guide*
- ″-qsmp Option″ in the *User's Guide*

DO SERIAL

Purpose

The **DO SERIAL** directive indicates to the compiler that the **DO** loop that is immediately following the directive must not be parallelized. This directive is useful in blocking automatic parallelization for a particular **DO** loop. The **DO SERIAL** directive only takes effect if you specify the **-qsmp** compiler option.

Format



Rules

The first noncomment line (not including other directives) that is following the **DO SERIAL** directive must be a **DO** loop. The **DO SERIAL** directive applies only to the **DO** loop that immediately follows the directive and not to any loops that are nested within that loop.

You can only specify one **DO SERIAL** directive for a given **DO** loop. The **DO SERIAL** directive must not appear with the **DO**, or **PARALLEL DO** directive on the same **DO** loop.

White space is optional between **DO** and **SERIAL.**

You should not use the OpenMP trigger constant with this directive.

Examples

Example 1: An example with nested DO loops where the inner loop (the J loop) is not parallelized.

```
!$OMP PARALLEL DO PRIVATE(S,I), SHARED(A)
      DO I=1, 500
        S=0
        !SMP$ DOSERIAL
        DO J=1, 500
          S=S+1
```

```
    ENDDO
      A(I)=S+I
  ENDDO
```

Example 2: An example with the DOSERIAL directive applied in nested loops. In this case, if automatic parallelization is enabled the I or K loop may be parallelized.

```
      DO I=1, 100
!SMP$ DOSERIAL
        DO J=1, 100
          DO K=1, 100
            ARR(I,J,K)=I+J+K
          ENDDO
        ENDDO
      ENDDO
```

Related Information

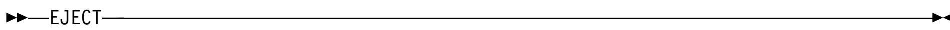
- "DO / END DO" on page 455
- "DO" on page 284
- "Loop Parallelization" on page 313
- "PARALLEL DO / END PARALLEL DO" on page 479
- ″-qdirective Option″ in the *User's Guide*
- ″-qsmp Option″ in the *User's Guide*

EJECT

Purpose

EJECT directs the compiler to start a new full page of the source listing. If there has been no source listing requested, the compiler will ignore this directive.

Format



Rules

The **EJECT** compiler directive can have an inline comment and a label. However, if you specify a statement label, the compiler discards it. Therefore, you must not reference any label on an **EJECT** directive. An example of using the directive would be placing it before a **DO** loop that you do not want split across pages in the listing. If you send the source listing to a printer, the **EJECT** directive provides a page break.

FLUSH

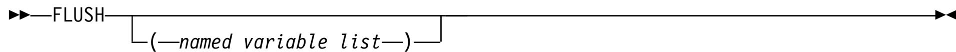
Purpose

The **FLUSH** directive ensures that each thread has access to data generated by other threads. This directive is required because the compiler may keep values in processor registers if a program is optimized. The **FLUSH** directive ensures that the memory images that each thread views are consistent.

The **FLUSH** directive only takes effect if you specify the **-qsmp** compiler option.

You might be able to improve the performance of your program by using the **FLUSH** directive instead of the **VOLATILE** attribute. The **VOLATILE** attribute causes variables to be flushed after every update and before every use, while **FLUSH** causes variables to be written to or read from memory only when specified.

Format



Rules

You can specify this directive anywhere in your code; however, if you specify it outside of the dynamic extent of a parallel region, it is ignored.

If you specify a *named_variable_list*, only the variables in that list are written to or read from memory (assuming that they have not been written or read already). All variables in the *named_variable_list* must be at the current scope and must be thread visible. Thread visible variables can be any of the following:

- Globally visible variables (common blocks and module data)
- Local and host-associated variables with the **SAVE** attribute
- Local variables without the **SAVE** attribute that are specified in a **SHARED** clause in a parallel region within the subprogram
- Local variables without the **SAVE** attribute that have had their addresses taken
- All pointer dereferences
- Dummy arguments

If you do not specify a *named_variable_list*, all thread visible variables are written to or read from memory.

When a thread encounters the **FLUSH** directive, it writes into memory the modifications to the affected variables. The thread also reads the latest copies

of the variables from memory if it has local copies of those variables: for example, if it has copies of the variables in registers.

It is not mandatory for all threads in a team to use the **FLUSH** directive. However, to guarantee that all thread visible variables are current, any thread that modifies a thread visible variable should use the **FLUSH** directive to update the value of that variable in memory. If you do not use **FLUSH** or one of the directives that implies **FLUSH** (see below), the value of the variable might not be the most recent one.

Note that **FLUSH** is not atomic. You must **FLUSH** shared variables that are controlled by a shared lock variable with one directive and then **FLUSH** the lock variable with another. This guarantees that the shared variables are written before the lock variable.

The following directives imply a **FLUSH** directive unless you specify a **NOWAIT** clause for those directives to which it applies:

- **BARRIER**
- **CRITICAL/END CRITICAL**
- **END DO**
- **END PARALLEL**
- **END SECTIONS**
- **END SINGLE**
- **ORDERED/END ORDERED**

Examples

Example 1: In the following example, two threads perform calculations in parallel and are synchronized when the calculations are complete:

```
      PROGRAM P
      INTEGER ISYNC(0:1), IAM

!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(ISYNC)
      IAM = OMP_GET_THREAD_NUM()
      ISYNC(IAM) = 0
!$OMP BARRIER
      CALL WORK
!$OMP FLUSH(ISYNC)
      ISYNC(IAM) = 1                         ! Each thread sets a flag
                                             ! once it has
!$OMP FLUSH(ISYNC)                           ! completed its work.
      DO WHILE (ISYNC(1-IAM) .eq. 0)         ! One thread waits for
                                             ! another to complete
!$OMP   FLUSH(ISYNC)                         ! its work.
      END DO

!$OMP END PARALLEL
```

```
      END PROGRAM P

      SUBROUTINE WORK                         ! Each thread does indep-
                                              ! endent calculations.
!       ...
!$OMP FLUSH                                   ! flush work variables
                                              ! before ISYNC
                                              ! is flushed.
      END SUBROUTINE WORK
```

Example 2: The following example is not valid, because it attempts to use **FLUSH** with a variable that is not thread visible:

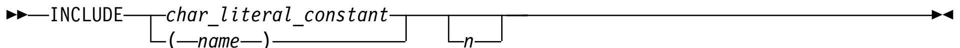
```
      FUNCTION F()
      INTEGER, AUTOMATIC :: i
!$OMP FLUSH(I)
      END FUNCTION F
```

INCLUDE

Purpose

The **INCLUDE** compiler directive inserts a specified statement or a group of statements into a program unit.

Format



name, *char_literal_constant (delimiters are optional)*
: specifies *filename*, the name of an include file

Under the AIX operating system, it need not specify the full path of the desired file, but it must specify the file extension if one exists.

name must contain only characters allowable in the XL Fortran character set. See "Characters" on page 13 for the character set supported by XL Fortran.

char_literal_constant is a character literal constant.

n
: is the value the compiler uses to decide whether to include the file during compilation. It can be any number from 1 through 255, and cannot specify a kind type parameter. If you specify *n*, the compiler includes the file only if the number appears as a suboption in the **-qci** (conditional include) compiler option. If you do not specify *n*, the compiler always includes the file.

A feature called conditional **INCLUDE** provides a means for selectively activating **INCLUDE** compiler directives within the Fortran source during compilation. You specify the included files by means of the **-qci** compiler option.

In fixed source form, the **INCLUDE** compiler directive must start after column 6, and can have a label.

You can add an inline comment to the **INCLUDE** line.

Rules

An included file can contain any complete Fortran source statements and compiler directives, including other **INCLUDE** compiler directives. Recursive **INCLUDE** compiler directives are not allowed. An **END** statement can be part of the included group. The first and last included lines must not be continuation lines. The statements in the include file are processed with the source form of the including file.

If the **SOURCEFORM** directive appears in an include file, the source form reverts to that of the including file once processing of the include file is complete. After the inclusion of all groups, the resulting Fortran program must follow all of the Fortran rules for statement order.

For an **INCLUDE** compiler directive with the left and right parentheses syntax, XL Fortran translates the file name to lowercase unless the **-qmixed** compiler option is on.

The AIX file system locates the file specified by *filename* as follows:
- If the first nonblank character of *filename* is `/`, *filename* specifies an absolute file name.
- If the first nonblank character is not `/`, the AIX operating system searches directories in order of decreasing priority:
  – If you specify any **-I** compiler option, *filename* is searched for in the directories specified.
  – If the AIX operating system cannot find *filename* then it searches :
    - the current directory for file *filename*.
    - the resident directory of the compiling source file for file *filename*.
    - directory `/usr/include` for file *filename*.

Examples

```
INCLUDE '/u/userid/dc101'     ! full absolute file name specified
INCLUDE '/u/userid/dc102.inc' ! INCLUDE file name has an extension
INCLUDE 'userid/dc103'        ! relative path name specified

INCLUDE (ABCdef)              ! includes file abcdef

INCLUDE '../Abc'              ! includes file Abc from parent directory
                              ! of directory being searched
```

Related Information

"**-qci** Option" in the *User's Guide*

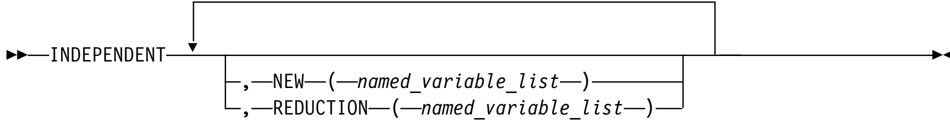
INDEPENDENT

Purpose

The **INDEPENDENT** directive, if used, must precede a **DO** loop, **FORALL** statement, or **FORALL** construct. It specifies that each operation in the **FORALL** statement or **FORALL** construct, can be executed in any order without affecting the semantics of the program. It also specifies each iteration of the **DO** loop, can be executed without affecting the semantics of the program.

This directive only takes effect if you specify either the **-qsmp** or **-qhot** compiler option.

Format



Rules

The first noncomment line (not including other directives) following the **INDEPENDENT** directive must be a **DO** loop, **FORALL** statement, or the first statement of a **FORALL** construct. This line cannot be an infinite **DO** or **DO WHILE** loop. The **INDEPENDENT** directive applies only to the **DO** loop that is immediately following the directive and not to any nested **DO** loops.

An **INDEPENDENT** directive can have at most one **NEW** clause and at most one **REDUCTION** clause.

If the directive applies to a **DO** loop, no iteration of the loop can interfere with any other iteration. Interference occurs in the following situations:

- Two operations that define, undefine, or redefine the same atomic object (data that has no subobjects) interfere, unless the parent object appears in the **NEW** clause or **REDUCTION** clause. You must define nested **DO** loop index variables in the **NEW** clause.
- Definition, undefinition, or redefinition of an atomic object interferes with any use of the value of the object. The exception is if the parent object appeared in the **NEW** clause or **REDUCTION** clause.

- Any operation that causes the association status of a pointer to become defined or undefined interferes with any reference to the pointer or any other operation that causes the association status to become defined or undefined.
- Transfer of control outside the **DO** loop or execution of an **EXIT**, **STOP**, or **PAUSE** statement interferes with all other iterations.
- Any two I/O operations associated with the same file or external unit interfere with each other. The exceptions to this rule are:
  – If the two I/O operations are two **INQUIRE** statements; or
  – If the two I/O operations are to distinct records of a direct access file.
- A change in the allocation status of an allocatable object between iterations causes interference.

If the **NEW** clause is specified, the directive must apply to a **DO** loop. The **NEW** clause modifies the directive and any surrounding **INDEPENDENT** directives by accepting any assertions made by such directive(s) as true. It does this *even if* the variables specified in the **NEW** clause are modified by each iteration of the loop. Variables specified in the **NEW** clause behave as if they are private to the body of the **DO** loop. That is, the program is unaffected if these variables (and any variables associated with them) were to become undefined both before and after each iteration of the loop.

Any variable you specify in the **NEW** clause or **REDUCTION** clause must not:
- Be a dummy argument
- Be a pointee
- Be use-associated or host-associated
- Be a common block variable
- Have either the **SAVE** or **STATIC** attribute
- Have either the **POINTER** or **TARGET** attribute
- Appear in an **EQUIVALENCE** statement

For **FORALL**, no combination of index values affected by the **INDEPENDENT** directive assigns to an atomic storage unit that is required by another combination. If a **DO** loop, **FORALL** statement, or **FORALL** construct all have the same body and each is preceded by an **INDEPENDENT** directive, they behave the same way.

The **REDUCTION** clause asserts that updates to named variables will occur within **REDUCTION** statements in the **INDEPENDENT** loop. Furthermore, the intermediate values of the **REDUCTION** variables are not used within the parallel section, other than in the updates themselves. Thus, the value of the **REDUCTION** variable after the construct is the result of a reduction tree.

If you specify the **REDUCTION** clause, the directive must apply to a **DO** loop. The only reference to a **REDUCTION** variable in an **INDEPENDENT DO** loop must be within a reduction statement.

A **REDUCTION** variable must be of intrinsic type, but must not be of type character. A **REDUCTION** variable must not be an allocatable array.

A **REDUCTION** variable must not occur in:
- A **NEW** clause in the same **INDEPENDENT** directive
- A **NEW** or **REDUCTION** clause in an **INDEPENDENT** directive in the body of the following **DO** loop
- A **FIRSTPRIVATE**, **PRIVATE** or **LASTPRIVATE** clause in a **PARALLEL DO** directive in the body of the following **DO** loop
- A **PRIVATE** clause in a **PARALLEL SECTIONS** directive in the body of the following **DO** loop

A **REDUCTION** statement can have one of the following forms:

```
►►──reduction_var_ref──=──expr──reduction_op──reduction_var_ref──────────────────►◄
```

```
►►──reduction_var_ref──=──reduction_var_ref──reduction_op──expr──────────────────►◄
```

```
►►──reduction_var_ref =──reduction_function──(expr,──reduction_var_ref)──────────►◄
```

```
►►──reduction_var_ref =──reduction_function──(reduction_var_ref,──expr)──────────►◄
```

where:

reduction_var_ref
: is a variable or subobject of a variable that appears in a **REDUCTION** clause

reduction_op
: is one of: **+**, **–**, *, **.AND.**, **.OR.**, **.EQV.**, **.NEQV.**, or **.XOR.**

reduction_function
: is one of: **MAX**, **MIN**, **IAND**, **IOR**, or **IEOR**

The following rules apply to **REDUCTION** statements:
1. A reduction statement is an assignment statement that occurs in the range of an **INDEPENDENT DO** loop. A variable in the **REDUCTION** clause must only occur in a **REDUCTION** statement within the **INDEPENDENT DO** loop.
2. The two *reduction_var_ref*s that appear in a **REDUCTION** statement must be lexically identical.

3. The syntax of the **INDEPENDENT** directive does not allow you to designate an array element or array section as a **REDUCTION** variable in the **REDUCTION** clause. Although such a subobject may occur in a **REDUCTION STATEMENT**, it is the entire array that is treated as a **REDUCTION** variable.
4. You cannot use the following form of the **REDUCTION** statement:

```
►►—reduction_var_ref— = —expr— - —reduction_var_ref——————————————►◄
```

Examples

Example 1:

```
      INTEGER A(10),B(10,12),F
!IBM* INDEPENDENT                     ! The NEW clause cannot be
      FORALL (I=1:9:2) A(I)=A(I+1)    ! specified before a FORALL
!IBM* INDEPENDENT, NEW(J)
      DO M=1,10
        J=F(M)                        ! 'J' is used as a scratch
        A(M)=J*J                      ! variable in the loop
!IBM*   INDEPENDENT, NEW(N)
        DO N=1,12                     ! The first executable statement
          B(M,N)=M+N*N                ! following the INDEPENDENT must
        END DO                        ! be either a DO or FORALL
      END DO
      END
```

Example 2:

```
      X=0
!IBM* INDEPENDENT, REDUCTION(X)
      DO J = 1, M
        X = X + J**2
      END DO
```

Example 3:

```
      INTEGER A(100), B(100, 100)
!SMP$ INDEPENDENT, REDUCTION(A), NEW(J)   ! Example showing an array used
      DO I=1,100                          ! for a reduction variable
        DO J=1, 100
          A(I)=A(I)+B(J, I)
        END DO
      END DO
```

Related Information

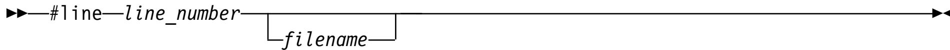
- "Loop Parallelization" on page 313
- "DO Construct" on page 126
- "FORALL" on page 311
- "-qdirective Option" in the *User's Guide*
- "-qhot Option" in the *User's Guide*
- "-qsmp Option" in the *User's Guide*

#line

Purpose

The **#line** directive associates code that is created by cpp or any other Fortran source code generator with input code created by the programmer. Because the preprocessor may cause lines of code to be inserted or deleted, the **#line** directive can be useful in error reporting and debugging because it identifies which lines in the original source caused the preprocessor to generate the corresponding lines in the intermediate file.

Format



The **#line** directive is a noncomment directive and follows the syntax rules for this type of directive.

line_number
: is a positive, unsigned integer literal constant without a **KIND** parameter. You must specify *line_number*.

filename
: is a character literal constant, with no kind type parameter. The *filename* may specify a full or relative path. The *filename* as specified will be recorded for use later. If you specify a relative path, when you debug the program the debugger will use its directory search list to resolve the *filename*.

Rules

The **#line** directive follows the same rules as other noncomment directives, with the following exceptions:

- You cannot have Inline comments on the same line as the **#line** directive.
- White space is optional between the **#** character and **line** in free source form.
- White space may not be embedded between the characters of the word **line** in fixed or free source forms.
- The **#line** directive can start anywhere on the line in fixed source form.

The **#line** directive indicates the origin of all code following the directive in the current file. Another **#line** directive will override a previous one.

If you supply a *filename*, the subsequent code in the current file will be as if it originated from that *filename*. If you omit the *filename*, and no previous **#line** directive with a specified *filename* exists in the current file, the code in the current file is treated as if it originated from the current file at the line

number specified. If a previous **#line** directive with a specified *filename* does exist in the current file, the *filename* from the previous directive is used.

line_number indicates the position, in the appropriate file, of the line of code following the directive. Subsequent lines in that file are assumed to have a one to one correspondence with subsequent lines in the source file until another **#line** directive is specified or the file ends.

When XL Fortran invokes cpp for a file, the preprocessor will emit **#line** directives unless you also specify the **-d** option.

Examples

The file test.F contains:

```
!  File test.F, Line 1
#include "test.h"
PRINT*, "test.F Line 3"
...
PRINT*, "test.F Line 6"
#include "test.h"
PRINT*, "test.F Line 8"
END
```

The file test.h contains:

```
!  File test.h line 1
RRINT*,1             ! Syntax Error
PRINT*,2
```

After the C preprocessor (**/lib/cpp**) processes the file test.F with the default options:

```
#line 1 "test.F"
! File test.F, Line 1
#line 1 "test.h"
! File test.h Line 1
RRINT*,1            ! Syntax Error
PRINT*,2
#line 3 "test.F"
PRINT*, "test.F Line 3"
...
#line 6
PRINT*, "test.F Line 6"
#line 1 "test.h"
!  File test.h Line 1
RRINT*,1            ! Syntax Error
PRINT*,2
#line 8 "test.F"
PRINT*, "test.F Line 8"
END
```

The compiler displays the following messages after it processes the file that is created by the C preprocessor:

```
2       2 |RRINT*,1           !Syntax error
            ......a................
a - "t.h", line 2.6: 1515-019 (S) Syntax is incorrect.

4       2 |RRINT*,1           !Syntax error
            ......a................
a - "t.h", line 2.6: 1515-019 (S) Syntax is incorrect.
```

Related Information

- "-d Option" in the *User's Guide*
- "Passing Fortran Files through the C Preprocessor" in the *User's Guide*
- "PARALLEL DO / END PARALLEL DO" on page 479
- "PARALLEL SECTIONS / END PARALLEL SECTIONS" on page 483

MASTER / END MASTER

Purpose

The **MASTER** construct enables you to define a block of code that will be run by only the master thread of the team. It includes a **MASTER** directive that precedes a block of code and ends with an **END MASTER** directive.

The **MASTER** and **END MASTER** directives only take effect if you specify the **-qsmp** compiler option.

Format

block represents the block of code that will be run by the master thread of the team.

Rules

It is illegal to branch into or out of a **MASTER** construct.

A **MASTER** directive binds to the closest dynamically enclosing **PARALLEL** directive, if one exists.

A **MASTER** directive cannot appear within the dynamic extent of a work-sharing construct or within the dynamic extent of the **PARALLEL DO**, and **PARALLEL SECTIONS** directives.

No implied barrier exists on entry to, or exit from, the **MASTER** construct.

Examples

Example 1: An example of the **MASTER** directive binding to the **PARALLEL** directive.

```
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP MASTER
         Y = 10.0
         X =  0.0
         DO I = 1, 4
             X = X + COS(Y) + I
         END DO
!$OMP END MASTER
!$OMP BARRIER
!$OMP DO PRIVATE(J)
         DO J = 1, 10000
              A(J) = X + SIN(J*2.5)
         END DO
!$OMP END DO
!$OMP END PARALLEL
         END
```

Related Information

- “Loop Parallelization” on page 313
- ″-qdirective Option″ in the *User's Guide*
- ″-qsmp Option″ in the *User's Guide*
- “PARALLEL DO / END PARALLEL DO” on page 479
- “PARALLEL SECTIONS / END PARALLEL SECTIONS” on page 483

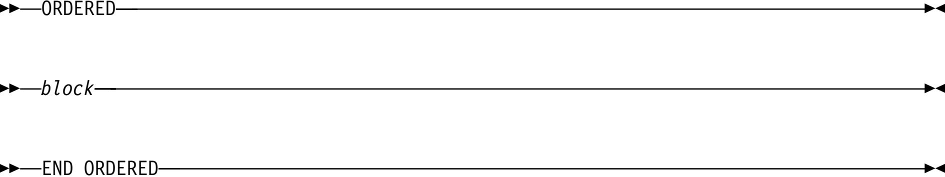
ORDERED / END ORDERED

Purpose

The **ORDERED / END ORDERED** directives cause the iterations of a block of code within a parallel loop to be executed in the order that the loop would execute in if it was run sequentially. You can force the code inside the **ORDERED** construct to run in a predictable order while code outside of the construct runs in parallel.

The **ORDERED** and **END ORDERED** directives only take effect if you specify the **-qsmp** compiler option.

Format



block represents the block of code that will be executed in sequence.

Rules

The **ORDERED** directive can only appear in the dynamic extent of a **DO** or **PARALLEL DO** directive. It is illegal to branch into or out of an **ORDERED** construct.

The **ORDERED** directive binds to the nearest dynamically enclosing **DO** or **PARALLEL DO** directive. You must specify the **ORDERED** clause on the **DO** or **PARALLEL DO** directive to which the **ORDERED** construct binds.

ORDERED constructs that bind to different **DO** directives are independent of each other.

Only one thread can execute an **ORDERED** construct at a time. Threads enter the **ORDERED** construct in the order of the loop iterations. A thread will enter the **ORDERED** construct if all of the previous iterations have either executed the construct or will never execute the construct.

Each iteration of a parallel loop with an **ORDERED** construct can only execute that **ORDERED** construct once. Each iteration of a parallel loop can execute at most one **ORDERED** directive. An **ORDERED** construct cannot appear within the dynamic extent of a **CRITICAL** construct.

Examples

Example 1: In this example, an **ORDERED** parallel loop counts down.

```
      PROGRAM P
!$OMP PARALLEL DO ORDERED
      DO I = 3, 1, -1
!$OMP ORDERED
      PRINT *,I
!$OMP END ORDERED
      END DO
      END PROGRAM P
```

The expected output of this program is:

```
3
2
1
```

Example 2: This example shows a program with two **ORDERED** constructs in a parallel loop. Each iteration can only execute a single section.

```
      PROGRAM P
!$OMP PARALLEL DO ORDERED
      DO I = 1, 3
        IF (MOD(I,2) == 0) THEN
!$OMP     ORDERED
            PRINT *, I*10
```

```
!$OMP     END ORDERED
        ELSE
!$OMP     ORDERED
            PRINT *, I
!$OMP     END ORDERED
        END IF
      END DO
      END PROGRAM P
```

The expected output of this program is:

```
1
20
3
```

Example 3: In this example, the program computes the sum of all elements of an array that are greater than a threshold. **ORDERED** is used to ensure that the results are always reproducible: roundoff will take place in the same order every time the program is executed, so the program will always produce the same results.

```
      PROGRAM P
        REAL :: A(1000)
        REAL :: THRESHOLD = 999.9
        REAL :: SUM = 0.0

!$OMP   PARALLEL DO ORDERED
        DO I = 1, 1000
          IF (A(I) > THRESHOLD) THEN
!$OMP       ORDERED
              SUM = SUM + A(I)
!$OMP       END ORDERED
          END IF
        END DO
      END PROGRAM P
```

Note: To avoid bottleneck situations when using the **ORDERED** clause, you can try using **DYNAMIC** scheduling or **STATIC** scheduling with a small chunk size. See “SCHEDULE” on page 491 for more information.

Related Information

- “Loop Parallelization” on page 313
- ″-qsmp Option″ in the *User’s Guide*
- “PARALLEL DO / END PARALLEL DO” on page 479
- “DO / END DO” on page 455
- “CRITICAL / END CRITICAL” on page 453
- “SCHEDULE” on page 491

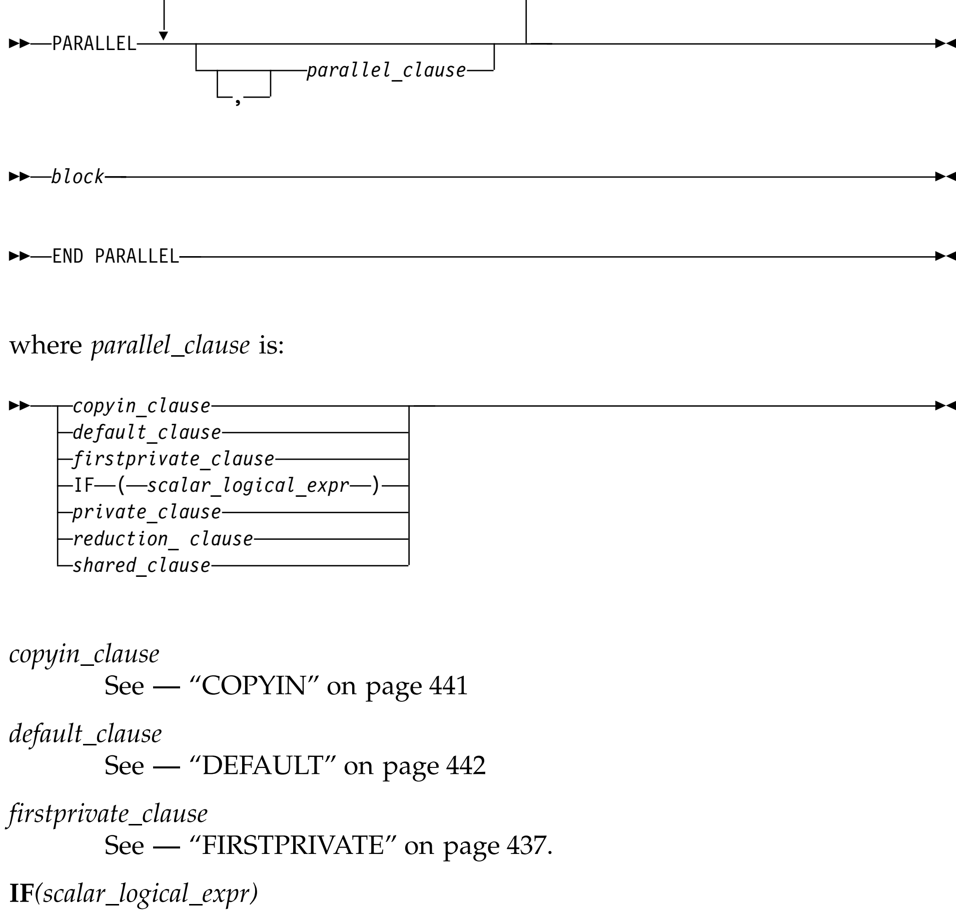
PARALLEL / END PARALLEL

Purpose

The **PARALLEL** construct enables you to define a block of code that can be executed by a team of threads concurrently. The **PARALLEL** construct includes a **PARALLEL** directive that is followed by one or more blocks of code, and ends with an **END PARALLEL** directive.

The **PARALLEL** and **END PARALLEL** directives only take effect if you specify the **-qsmp** compiler option.

Format



where *parallel_clause* is:

copyin_clause
: See — "COPYIN" on page 441

default_clause
: See — "DEFAULT" on page 442

firstprivate_clause
: See — "FIRSTPRIVATE" on page 437.

IF(*scalar_logical_expr*)
: If you specify the **IF** clause, the run-time environment performs a test to determine whether to run the block in serial or parallel. If *scalar_logical_expr* is true, then the block is run in parallel; if not, then the block is run in serial.

private_clause
: See — “PRIVATE” on page 436.

reduction_clause
: See — “REDUCTION” on page 438

shared_clause
: See — “SHARED” on page 443

Rules

It is illegal to branch into or out of a **PARALLEL** construct.

The **IF** and **DEFAULT** clauses can appear at most once in a **PARALLEL** directive.

An **IF** expression is evaluated outside of the context of the parallel construct. Any function reference in the **IF** expression must not have side effects.

A variable that appears in the **REDUCTION** clause of an enclosing **PARALLEL** construct cannot appear in the **FIRSTPRIVATE**, **LASTPRIVATE**, or **PRIVATE** clause of an enclosing **PARALLEL** construct, and cannot be made **PRIVATE** by using the **DEFAULT** clause in the inner **PARALLEL** construct.

You should be careful when you perform input/output operations in a parallel region. If multiple threads execute a Fortran I/O statement on the same unit, you should make sure that the threads are synchronized. If you do not, the behaviour is undefined. Also note that although in the XL Fortran implementation each thread has exclusive access to the I/O unit, the OpenMP specification does not require exclusive access.

Directives that bind to a parallel region will bind to that parallel region even if it is serialized.

The **END PARALLEL** directive implies the **FLUSH** directive.

Examples

Example 1: An example of an inner **PARALLEL** directive with the **PRIVATE** clause enclosing the **PARALLEL** construct. Note: The **SHARED** clause is present on the inner **PARALLEL** construct.

```
!$OMP PARALLEL PRIVATE(X)
!$OMP DO
          DO I = 1, 10
            X(I) = I
!$OMP PARALLEL SHARED (X,Y)
!$OMP DO
      DO K = 1, 10
         Y(K,I)= K * X(I)
      END DO
```

```
!$OMP END DO
!$OMP END PARALLEL
      END DO
!$OMP END DO
!$OMP END PARALLEL
```

Example 2: An example showing that a variable cannot appear in both a **PRIVATE**, and **SHARED** clause.

```
!$OMP PARALLEL PRIVATE(A), SHARED(A)
!$OMP DO
      DO I = 1, 1000
         A(I) = I * I
      END DO
!$OMP END DO
!$OMP END PARALLEL
```

Example 3: This example demonstrates the use of the **COPYIN** clause. Each thread created by the **PARALLEL** directive has its own copy of the common block **BLOCK**. The **COPYIN** clause causes the initial value of *FCTR* to be copied into the threads that execute iterations of the **DO** loop.

```
      PROGRAM TT
      COMMON /BLOCK/ FCTR
      INTEGER :: I, FCTR
!$OMP THREADPRIVATE(/BLOCK/)
      INTEGER :: A(100)

      FCTR = -1
      A = 0

!$OMP PARALLEL COPYIN(FCTR)
!$OMP DO
      DO I=1, 100
         FCTR = FCTR + I
         CALL SUB(A(I), I)
      ENDDO
!$OMP END PARALLEL

      PRINT *, A
      END PROGRAM

      SUBROUTINE SUB(AA, J)
      INTEGER :: FCTR, AA, J
      COMMON /BLOCK/ FCTR
!$OMP THREADPRIVATE(/BLOCK/)    ! EACH THREAD GETS ITS OWN COPY
                                ! OF BLOCK.
      AA = FCTR
      FCTR = FCTR - J
      END SUBROUTINE SUB
```

The expected output is:

```
0 1 2 3 ... 96 97 98 99
```

Related Information

- “FLUSH” on page 462
- “PARALLEL DO / END PARALLEL DO”
- “INDEPENDENT” on page 466
- “THREADPRIVATE” on page 510
- “DO / END DO” on page 455
- ″-qdirective Option″ in the *User's Guide*
- ″-qsmp Option″ in the *User's Guide*

PARALLEL DO / END PARALLEL DO

Purpose

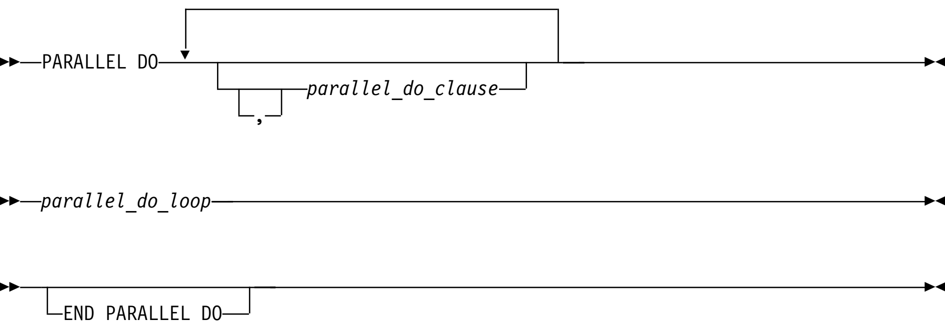
The **PARALLEL DO** directive enables you to specify which loops the compiler should parallelize. This is semantically equivalent to:

```
!$OMP PARALLEL
!$OMP DO
 ...
!$OMP ENDDO
!$OMP END PARALLEL
```

and is a convenient way of parallelizing loops. The **END PARALLEL DO** directive allows you to indicate the end of a **DO** loop that is specified by the **PARALLEL DO** directive.

The **PARALLEL DO** and **END PARALLEL DO** directives only take effect if you specify the **-qsmp** compiler option.

Format



where *parallel_do_clause* is:

```
>>-+-copyin_clause-------------------------------+-------------><
   +-default_clause------------------------------+
   +-firstprivate_clause-------------------------+
   +-IF--(--scalar_logical_expr--)---------------+
   +-lastprivate_clause--------------------------+
   +-ORDERED-------------------------------------+
   +-private_clause------------------------------+
   +-reduction_clause----------------------------+
   +-SCHEDULE--(--sched_type--+------+--)--------+
   |                          '-,n---'           |
   '-shared_clause-------------------------------'
```

copyin_clause
: See — "COPYIN" on page 441

default_clause
: See — "DEFAULT" on page 442

firstprivate_clause
: See — "FIRSTPRIVATE" on page 437.

IF*(scalar_logical_expr)*

If you specify the **IF** clause, the run-time environment performs a test to determine whether to run the block in serial or parallel. If *scalar_logical_expr* is true, then the block is run in parallel; if not, then the block is run in serial.

lastprivate_clause
: See — "LASTPRIVATE" on page 437.

ORDERED
: The parallel loop may contain **ORDERED** sections in its dynamic extent.

private_clause
: See — "PRIVATE" on page 436

reduction_clause
: See — "REDUCTION" on page 438

SCHEDULE*(sched_type[,n])*

sched_type
: is one of **AFFINITY**, **DYNAMIC**, **GUIDED**, **RUNTIME**, or **STATIC**

n
: must be a positive scalar integer expression; you must not specify it for the **RUNTIME** *sched_type*. See "SCHEDULE" on page 491

page 491 for definitions of these scheduling types. If you are using the *trigger_constant* **$OMP**, you should not specify the scheduling type **AFFINITY**.

shared_clause
See — "SHARED" on page 443

Rules

The first noncomment line (not including other directives) that is following the **PARALLEL DO** directive must be a **DO** loop. This line cannot be an infinite **DO** or **DO WHILE** loop. The **PARALLEL DO** directive applies only to the **DO** loop that is immediately following the directive, and not to any nested **DO** loops.

If you specify a **DO** loop by a **PARALLEL DO** directive, the **END PARALLEL DO** directive is optional. If you use the **END PARALLEL DO** directive, it must immediately follow the end of the **DO** loop.

You may have a **DO** construct that contains several **DO** statements. If the **DO** statements share the same **DO** termination statement, and an **END PARALLEL DO** directive follows the construct, you can only specify a **PARALLEL DO** directive for the outermost **DO** statement of the construct.

You must not follow the **PARALLEL DO** directive by a **DO** (work-sharing) or **DO SERIAL** directive. You can specify only one **PARALLEL DO** directive for a given **DO** loop.

All work-sharing constructs and **BARRIER** directives that are encountered must be encountered in the same order by all threads in the team.

The **PARALLEL DO** directive must not appear with the **INDEPENDENT** directive for a given **DO** loop.

Note: The **INDEPENDENT** directive allows you to keep your code common with **HPF** implementations. You should use the **PARALLEL DO** directive for maximum portability across multiple vendors. The **PARALLEL DO** directive is a prescriptive directive, while the **INDEPENDENT** directive is an assertion about the characteristics of the loop. For more information on the **INDEPENDENT** directive, see page 466.

The **IF** clause may appear at most once in a **PARALLEL DO** directive.

An **IF** expression is evaluated outside of the context of the parallel construct. Any function reference in the **IF** expression must not have side effects.

By default, a nested parallel loop is serialized, regardless of the setting of the **IF** clause. You can change this default by using the **-qsmp=nested_par** compiler option.

The **SCHEDULE** clause may appear at most once in a **PARALLEL DO** directive.

If the **REDUCTION** variable of an inner **DO** loop appears in the **PRIVATE** or **LASTPRIVATE** clause of an enclosing **DO** loop or **PARALLEL SECTIONS** construct, the variable must be initialized before the inner **DO** loop.

A variable that appears in the **REDUCTION** clause of an **INDEPENDENT** directive of an enclosing **DO** loop must not also appear in the *data_scope_entity_list* of the **PRIVATE** or **LASTPRIVATE** clause.

You should be careful when you perform input/output operations in a parallel region. If multiple threads execute a Fortran I/O statement on the same unit, you should make sure that the threads are synchronized. If you do not, the behaviour is undefined. Also note that although in the XL Fortran implementation each thread has exclusive access to the I/O unit, the OpenMP specification does not require exclusive access.

Directives that bind to a parallel region will bind to that parallel region even if it is serialized.

Examples

Example 1: A valid example with the **LASTPRIVATE** clause.

```
!$OMP PARALLEL DO PRIVATE(I), LASTPRIVATE (X)
      DO I = 1,10
        X = I * I
        A(I) = X * B(I)
      END DO
      PRINT *, X                     ! X has the value 100
```

Example 2: A valid example with the **REDUCTION** clause.

```
!$OMP PARALLEL DO PRIVATE(I), REDUCTION(+:MYSUM)
      DO I = 1, 10
        MYSUM = MYSUM + IARR(I)
      END DO
```

Example 3: A valid example where more than one thread accesses a variable that is marked as **SHARED**, but the variable is used only in a **CRITICAL** construct.

```
!$OMP PARALLEL DO SHARED (X)
      DO I = 1, 10
        A(I) = A(I) * I
```

```
!$OMP   CRITICAL
          X = X + A(I)
!$OMP   END CRITICAL
      END DO
```

Example 4: A valid example of the **END PARALLEL DO** directive.

```
         REAL A(100), B(2:100), C(100)
!$OMP PARALLEL DO
         DO I = 2, 100
             B(I) = (A(I) + A(I-1))/2.0
         END DO
!$OMP END PARALLEL DO
!$OMP PARALLEL DO
         DO  J = 1, 100
            C(J) = X + COS(J*5.5)
         END DO
!$OMP END PARALLEL DO
         END
```

Related Information

- “Loop Parallelization” on page 313
- ″-qdirective Option″ in the *User's Guide*
- ″-qsmp Option″ in the *User's Guide*
- “DO” on page 284
- “DO / END DO” on page 455
- “INDEPENDENT” on page 466
- “Loop Parallelization” on page 313
- “ORDERED / END ORDERED” on page 473
- “PARALLEL / END PARALLEL” on page 476
- “PARALLEL SECTIONS / END PARALLEL SECTIONS”
- “SCHEDULE” on page 491
- “THREADPRIVATE” on page 510

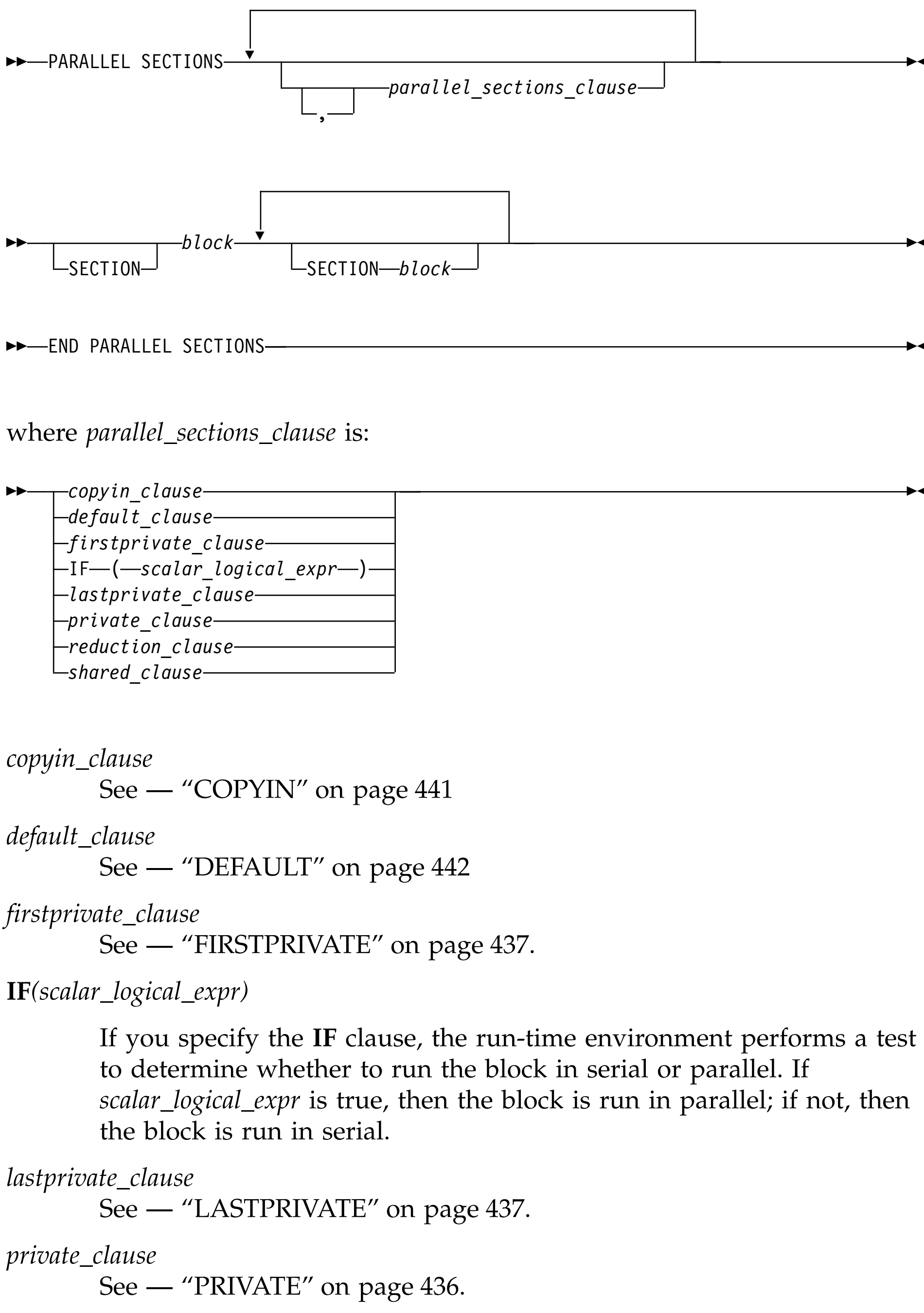
PARALLEL SECTIONS / END PARALLEL SECTIONS

Purpose

The **PARALLEL SECTIONS** construct enables you to define independent blocks of code that the compiler can execute concurrently. The **PARALLEL SECTIONS** construct includes a **PARALLEL SECTIONS** directive followed by one or more blocks of code delimited by the **SECTION** directive, and ends with an **END PARALLEL SECTIONS** directive.

The **PARALLEL SECTIONS**, **SECTION** and **END PARALLEL SECTIONS** directives only take effect if you specify the **-qsmp** compiler option.

Format



where *parallel_sections_clause* is:

copyin_clause
: See — "COPYIN" on page 441

default_clause
: See — "DEFAULT" on page 442

firstprivate_clause
: See — "FIRSTPRIVATE" on page 437.

IF*(scalar_logical_expr)*
: If you specify the **IF** clause, the run-time environment performs a test to determine whether to run the block in serial or parallel. If *scalar_logical_expr* is true, then the block is run in parallel; if not, then the block is run in serial.

lastprivate_clause
: See — "LASTPRIVATE" on page 437.

private_clause
: See — "PRIVATE" on page 436.

reduction_clause
See — “REDUCTION” on page 438

shared_clause
See — “SHARED” on page 443

Rules

The **PARALLEL SECTIONS** construct includes the delimiting directives, and the blocks of code they enclose. The rules below also refer to *sections*. You define a section as the block of code within the delimiting directives.

The **SECTION** directive marks the beginning of a block of code. At least one **SECTION** and its block of code must appear within the **PARALLEL SECTIONS** construct. Note, however, that you do not have to specify the **SECTION** directive for the first section. The end of a block is delimited by either another **SECTION** directive or by the **END PARALLEL SECTIONS** directive.

You can use the **PARALLEL SECTIONS** construct to specify parallel execution of the identified sections of code. There is no assumption as to the order in which sections are executed. Each section must not interfere with any other section in the construct unless the interference occurs within a **CRITICAL** construct. For the definition of interference outside a **CRITICAL** construct, see page 466.

It is illegal to branch into or out of any block of code that is defined by the **PARALLEL SECTIONS** construct.

The compiler determines how to divide the work among the threads based on a number of factors, such as the number of threads and the number of sections to be executed in parallel. Therefore, a single thread may execute more than one **SECTION**, or a thread may not execute any **SECTION**.

All work-sharing constructs and **BARRIER** directives that are encountered must be encountered in the same order by all threads in the team.

Within a **PARALLEL SECTIONS** construct, variables that are not appearing in the **PRIVATE** clause are assumed to be **SHARED** by default.

The **IF** clause may appear at most once in the a **PARALLEL SECTIONS** directive.

An **IF** expression is evaluated outside of the context of the parallel construct. Any function reference in the **IF** expression must not have side effects.

By default, a nested parallel loop is serialized, regardless of the setting of the **IF** clause. You can change this default by using the **-qsmp=nested_par** compiler option.

In a **PARALLEL SECTIONS** construct, a variable that appears in the **REDUCTION** clause of an **INDEPENDENT** directive or the **PARALLEL DO** directive of an enclosing **DO** loop must not also appear in the *data_scope_entity_list* of the **PRIVATE** clause.

If the **REDUCTION** variable of the inner **PARALLEL SECTIONS** construct appears in the **PRIVATE** clause of an enclosing **DO** loop or **PARALLEL SECTIONS** construct, the variable must be initialized before the inner **PARALLEL SECTIONS** construct.

The **PARALLEL SECTIONS** construct must not appear within a **CRITICAL** construct.

You should be careful when you perform input/output operations in a parallel region. If multiple threads execute a Fortran I/O statement on the same unit, you should make sure that the threads are synchronized. If you do not, the behaviour is undefined. Also note that although in the XL Fortran implementation each thread has exclusive access to the I/O unit, the OpenMP specification does not require exclusive access.

Directives that bind to a parallel region will bind to that parallel region even if it is serialized.

The **END PARALLEL SECTIONS** directive implies the **FLUSH** directive.

Examples

Example 1:

```
!$OMP PARALLEL SECTIONS
!$OMP   SECTION
          DO I = 1, 10
            C(I) = MAX(A(I), A(I+1))
          END DO
!$OMP   SECTION
          W = U + V
          Z = X + Y
!$OMP END PARALLEL SECTIONS
```

Example 2: In this example, the index variable I is declared as **PRIVATE**. Note also that the first optional **SECTION** directive has been omitted.

```
!$OMP PARALLEL SECTIONS PRIVATE(I)
          DO I = 1, 100
            A(I) = A(I) * I
          END DO
!$OMP   SECTION
```

```
          CALL NORMALIZE (B)
          DO I = 1, 100
            B(I) = B(I) + 1.0
          END DO
!$OMP   SECTION
          DO I = 1, 100
            C(I) = C(I) * C(I)
          END DO
!$OMP END PARALLEL SECTIONS
```

Example 3: This example is invalid because there is a data dependency for the variable C across sections.

```
!$OMP PARALLEL SECTIONS
!$OMP   SECTION
          DO I = 1, 10
            C(I) = C(I) * I
          END DO
!$OMP   SECTION
          DO K = 1, 10
            D(K) = C(K) + K
          END DO
!$OMP END PARALLEL SECTIONS
```

Related Information

- “PARALLEL / END PARALLEL” on page 476
- “PARALLEL DO / END PARALLEL DO” on page 479
- “INDEPENDENT” on page 466
- “THREADPRIVATE” on page 510
- "-qdirective Option" in the *User's Guide*
- "-qsmp Option" in the *User's Guide*

PERMUTATION

Purpose

The **PERMUTATION** directive specifies that the elements of each array that is listed in the *integer_array_name_list* have no repeated values. This directive is useful when you use array elements as subscripts for other array references.

The **PERMUTATION** directive only takes effect if you specify either the **-qsmp** or **-qhot** compiler option.

Format

```
►►──PERMUTATION──(──integer_array_name_list──)─────────────────────────►◄
```

integer_array_name
: is an integer array with no repeated values.

Rules

The first noncomment line (not including other directives) that is following the **PERMUTATION** directive must be a **DO** loop. This line cannot be an infinite **DO** or **DO WHILE** loop. The **PERMUTATION** directive applies only to the **DO** loop that is immediately following the directive, and not to any nested **DO** loops.

Examples

```
PROGRAM EX3
  INTEGER A(100), B(100)
  !SMP$  PERMUTATION (A)
  DO I = 1, 100
    A(I) = I
    B(A(I)) = B(A(I)) + A(I)
  END DO
END PROGRAM EX3
```

Related Information

- ″-qhot Option″ in the *User's Guide*
- ″-qsmp Option″ in the *User's Guide*
- “DO” on page 284

PREFETCH_BY_LOAD / PREFETCH_FOR_LOAD / PREFETCH_FOR_STORE

Purpose

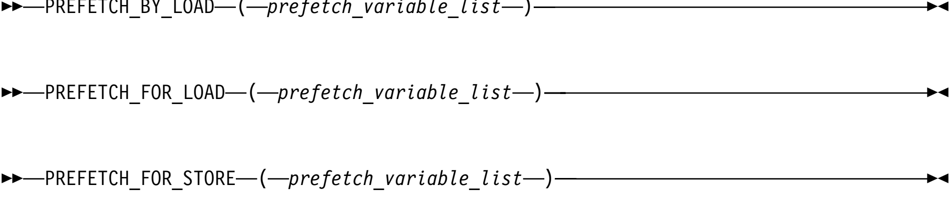
You can use data prefetching to instruct the compiler to load specific data from main memory into the cache before the data is referenced. Some prefetching may be done automatically by POWER3 hardware, but since compiler-assisted software prefetching uses information found in source code, using the directive significantly reduces the number of cache misses.

XL Fortran provides three directives for compiler-assisted software prefetching, as follows:

- The **PREFETCH_BY_LOAD** directive prefetches data into the cache by way of a load instruction. **PREFETCH_BY_LOAD** can be used on any machine, but if you are running on a POWER3 machine, **PREFETCH_BY_LOAD** enables hardware-assisted prefetching.
- The **PREFETCH_FOR_LOAD** directive prefetches data into the cache for reading by way of a cache prefetch instruction. The **PREFETCH_FOR_LOAD** directive only takes effect on a PowerPC machine.
- The **PREFETCH_FOR_STORE** directive prefetches data into the cache for writing by way of a cache prefetch instruction. The **PREFETCH_FOR_LOAD** directive only takes effect on a PowerPC machine.

PREFETCH_BY_LOAD / PREFETCH_FOR_LOAD / PREFETCH_FOR_STORE

Format



prefetch_variable
: is a variable to be prefetched. The variable cannot be a zero-sized array, a zero-sized substring, or an array section with a vector subscript.

Rules

To use the **PREFETCH_BY_LOAD**, **PREFETCH_FOR_LOAD**, and **PREFETCH_FOR_STORE** directives, you must specify the **-O** option at level 2 or greater. In addition, for the **PREFETCH_FOR_LOAD** and **PREFETCH_FOR_STORE** directives, you must compile for a PowerPC machine.

When you prefetch a variable, the memory block that includes the variable address is loaded into the cache. A memory block is equal to the size of a cache line. Since the variable you are loading into the cache may appear anywhere within the memory block, you may not be able to prefetch all the elements of an array.

These directives may appear anywhere in your source code where executable constructs may appear.

These directives can add run-time overhead to your program. Therefore you should use the directives only where necessary.

Examples

This example shows valid uses of the **PREFETCH_BY_LOAD**, **PREFETCH_FOR_LOAD**, and **PREFETCH_FOR_STORE** directives.

For this example, assume that the size of the cache line is 64 bytes and that none of the declared data items exist in the cache at the beginning of the program. The rationale for using the directives is as follows:

- All elements of array *ARRA* will be assigned; therefore, you can use the **PREFETCH_FOR_STORE** directive to bring the first 16 and second 16 elements of the array into the cache before they are referenced.
- Since all elements of array *ARRC* will be read, you can use the **PREFETCH_FOR_LOAD** directive to bring the first 16 and second 16

elements of the array into the cache before they are referenced. (Assume that the elements have been initialized first.)

- Each iteration of the loop will use variables *A*, *B*, *C*, *TEMP*, *I*, *K* and array element *ARRB(I*32)*; you can use the **PREFETCH_BY_LOAD** directive to load the variables and the array into the cache. (Because of the size of the cache line, you will fetch 16 elements of *ARRB*, starting at element *ARRB(I*32)*).

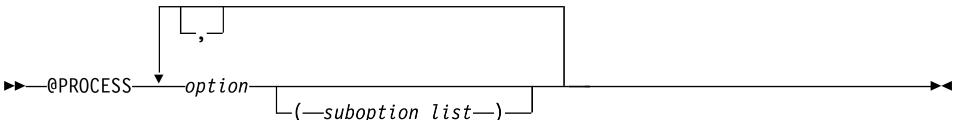
```
      PROGRAM GOODPREFETCH

      REAL*4 A, B, C, TEMP
      REAL*4 ARRA(2**5), ARRB(2**10), ARRC(2**5)
      INTEGER(4) I, K

! Bring ARRA into cache for writing.
!IBM* PREFETCH_FOR_STORE (ARRA(1), ARRA(2**4+1))

! Bring ARRC into cache for reading.
!IBM* PREFETCH_FOR_LOAD (ARRC(1), ARRC(2**4+1))

! Bring all variables into the cache.
!IBM* PREFETCH_BY_LOAD (A, B, C, TEMP, I , K)

! A subroutine is called to allow clock cycles to pass so that the
! data is loaded into the cache before the data is referenced.
      CALL FOO()
      K = 32
      DO I = 1, 2 ** 10

! Bring ARRB(I*K) into the cache
!IBM* PREFETCH_BY_LOAD (ARRB(I*K))
        A = -I
        B = I + 1
        C = I + 2
        TEMP = SQRT(B*B - 4*A*C)
        ARRA(I) = ARRC(I) + (-B + TEMP) / (2*A)
        ARRB(I*K) = (-B - TEMP) / (2*A)
      END DO
      END PROGRAM GOODPREFETCH
```

@PROCESS

Purpose

You can specify compiler options to affect an individual compilation unit by putting the **@PROCESS** compiler directive in the source file. It can override options that are specified in the configuration file, in the default settings, or on the command line.

Format



option is the name of a compiler option, without the **-q**

suboption
is a suboption of a compiler option

Rules

In fixed source form, **@PROCESS** can start in column 1 or after column 6. In free source form, the **@PROCESS** compiler directive can start in any column.

You cannot place a statement label or inline comment on the same line as an **@PROCESS** compiler directive.

By default, any option settings you designate with the **@PROCESS** compiler directive are effective only for the compilation unit in which the statement appears. If the file has more than one compilation unit, the option setting is reset to its original state before the next unit is compiled. Trigger constants specified by the **DIRECTIVE** option are in effect until the end of the file (or until **NODIRECTIVE** is processed).

The **@PROCESS** compiler directive must usually appear before the first statement of a compilation unit. The only exceptions are for **SOURCE** and **NOSOURCE**, which you can put in **@PROCESS** directives anywhere in the compilation unit.

Related Information

See the *User's Guide* for details on compiler options.

SCHEDULE

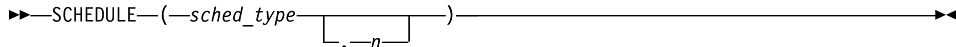
Purpose

The **SCHEDULE** directive allows 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.

The **SCHEDULE** directive only takes effect if you specify the **-qsmp** compiler option.

SCHEDULE

Format



n
: *n* must be a positive, specification expression. You must not specify *n* for the *sched_type* **RUNTIME**.

sched_type
: is **AFFINITY**, **DYNAMIC**, **GUIDED**, **RUNTIME**, or **STATIC**

Definitions:

number_of_iterations
: is the number of iterations in the loop to be parallelized.

number_of_threads
: is the number of threads used by the program.

AFFINITY
The iterations of a loop are initially divided into *number_of_threads* partitions, containing

```
CEILING(number_of_iterations / number_of_threads)
```

iterations. Each partition is initially assigned to a thread, and is then further subdivided into chunks containing *n* iterations, if *n* has been specified. If *n* has not been specified, then the chunks consist of

```
CEILING(number_of_iterations_remaining_in_partition / 2)
```

loop iterations.

When a thread becomes free, it takes the next chunk from its initially assigned partition. If there are no more chunks in that partition, then the thread takes the next available chunk from a partition that is initially assigned to another thread.

Threads that are active will complete the work in a partition that is initially assigned to a sleeping thread.

DYNAMIC
If *n* has been specified, the iterations of a loop are divided into chunks containing *n* iterations each. If *n* has not been specified, then the default chunk size is 1 iteration.

Threads are assigned these chunks on a "first-come, first-do" basis. Chunks of the remaining work are assigned to available threads, until all work has been assigned.

If a thread is asleep, its assigned work will be taken over by an active thread, once that other thread becomes available.

GUIDED

If you specify a value for *n*, the iterations of a loop are divided into chunks such that the size of each successive chunk is exponentially decreasing. *n* specifies the size of the smallest chunk, except possibly the last. If you do not specify a value for *n*, the default value is 1.

The size of the initial chunk is

```
CEILING(number_of_iterations / number_of_threads)
```

iterations. Subsequent chunks consist of

```
CEILING(number_of_iterations_remaining / number_of_threads)
```

iterations. As each thread finishes a chunk, it dynamically obtains the next available chunk.

You can use guided scheduling in a situation in which multiple threads in a team might arrive at a **DO** work-sharing construct at varying times, and each iteration requires roughly the same amount of work. For example, if you have a **DO** loop preceded by one or more work-sharing **SECTIONS** or **DO** constructs with **NOWAIT** clauses, you can guarantee that no thread waits at the barrier longer than it takes another thread to execute its final iteration, or final k iterations if a chunk size of k is specified. The **GUIDED** schedule requires the fewest synchronizations of all the scheduling methods.

An *n* expression is evaluated outside of the context of the **DO** construct. Any function reference in the *n* expression must not have side effects.

The value of the *n* parameter on the **SCHEDULE** clause must be the same for all of the threads in the team.

RUNTIME

Determine the scheduling type at run time.

At run time, the scheduling type can be specified using the environment variable **XLSMPOPTS**. If no scheduling type is specified using that variable, then the default scheduling type used is **STATIC**.

STATIC

If *n* has been specified, the iterations of a loop are divided into chunks that contain *n* iterations. Each thread is assigned chunks in a ″round robin″ fashion. This is known as block cyclic scheduling. If the value of *n* is 1, then the scheduling type is specifically referred to as cyclic scheduling.

If *n* has not been specified, the chunks will contain

```
CEILING(number_of_iterations / number_of_threads)
```

iterations. Each thread is assigned one of these chunks. This is known as block cyclic scheduling.

If a thread is asleep and it has been assigned work, it will be awakened so that it may complete its work.

STATIC is the default scheduling type if the user has not specified any scheduling type at compile-time or run time.

Rules

The **SCHEDULE** directive must appear in the specification part of a scoping unit.

Only one **SCHEDULE** directive may appear in the specification part of a scoping unit.

The **SCHEDULE** directive applies to one of the following:

- All loops in the scoping unit that do not already have explicit scheduling types specified. Individual loops can have scheduling types specified using the **SCHEDULE** clause of the **PARALLEL DO** directive.
- Loops that the compiler generates and have been chosen to be parallelized by automatic parallelization. For example, the **SCHEDULE** directive applies to loops generated for **FORALL**, **WHERE**, I/O implied-**DO**, and array constructor implied-**DO**.

Any dummy arguments appearing or referenced in the specification expression for the chunk size *n* must also appear in the **SUBROUTINE** or **FUNCTION** statement and in all **ENTRY** statements appearing in the given subprogram.

If the specified chunk size *n* is greater than the number of iterations, the loop will not be parallelized and will execute on a single thread.

If you specify more than one method of determining the chunking algorithm, the compiler will follow, in order of precedence:

1. **SCHEDULE** clause to the **PARALLEL DO** directive.
2. **SCHEDULE** directive
3. **schedule** suboption to the **-qsmp** compiler option. See ″-qsmp Option″ in the *User's Guide*
4. **XLSMPOPTS** run-time option. See ″XLSMPOPTS″ in the *User's Guide*
5. run-time default (that is, **STATIC**)

Examples

Example 1. Given the following information:

```
number of iterations = 1000
number of threads = 4
```

and using the **GUIDED** scheduling type, the chunk sizes would be as follows:

```
250 188 141 106 79 59 45 33 25 19 14 11 8 6 4 3 3 2 1 1 1 1
```

The iterations would then be divided into the following chunks:

```
chunk  1 = iterations    1 to  250
chunk  2 = iterations  251 to  438
chunk  3 = iterations  439 to  579
chunk  4 = iterations  580 to  685
chunk  5 = iterations  686 to  764
chunk  6 = iterations  765 to  823
chunk  7 = iterations  824 to  868
chunk  8 = iterations  869 to  901
chunk  9 = iterations  902 to  926
chunk 10 = iterations  927 to  945
chunk 11 = iterations  946 to  959
chunk 12 = iterations  960 to  970
chunk 13 = iterations  971 to  978
chunk 14 = iterations  979 to  984
chunk 15 = iterations  985 to  988
chunk 16 = iterations  989 to  991
chunk 17 = iterations  992 to  994
chunk 18 = iterations  995 to  996
chunk 19 = iterations  997 to  997
chunk 20 = iterations  998 to  998
chunk 21 = iterations  999 to  999
chunk 22 = iterations 1000 to 1000
```

A possible scenario for the division of work could be:

```
thread 1 executes chunks 1 5 10 13 18 20
thread 2 executes chunks 2 7  9 14 16 22
thread 3 executes chunks 3 6 12 15 19
thread 4 executes chunks 4 8 11 17 21
```

Example 2. Given the following information:

```
number of iterations = 100
number of threads = 4
```

and using the **AFFINITY** scheduling type, the iterations would be divided into the following partitions:

```
partition 1 = iterations  1 to  25
partition 2 = iterations 26 to  50
partition 3 = iterations 51 to  75
partition 4 = iterations 76 to 100
```

The partitions would be divided into the following chunks:

```
chunk 1a = iterations   1 to  13
chunk 1b = iterations  14 to  19
chunk 1c = iterations  20 to  22
chunk 1d = iterations  23 to  24
chunk 1e = iterations  25 to  25

chunk 2a = iterations  26 to  38
chunk 2b = iterations  39 to  44
chunk 2c = iterations  45 to  47
chunk 2d = iterations  48 to  49
chunk 2e = iterations  50 to  50

chunk 3a = iterations  51 to  63
chunk 3b = iterations  64 to  69
chunk 3c = iterations  70 to  72
chunk 3d = iterations  73 to  74
chunk 3e = iterations  75 to  75

chunk 4a = iterations  76 to  88
chunk 4b = iterations  89 to  94
chunk 4c = iterations  95 to  97
chunk 4d = iterations  98 to  99
chunk 4e = iterations 100 to 100
```

A possible scenario for the division of work could be:

```
thread 1 executes chunks 1a 1b 1c 1d 1e 4d
thread 2 executes chunks 2a 2b 2c 2d
thread 3 executes chunks 3a 3b 3c 3d 3e 2e
thread 4 executes chunks 4a 4b 4c 4e
```

Note that in this scenario, thread 1 finished executing all the chunks in its partition and then grabbed an available chunk from the partition of thread 4. Similarly, thread 3 finished executing all the chunks in its partition and then grabbed an available chunk from the partition of thread 2.

Example 3. Given the following information:

```
number of iterations = 1000
number of threads = 4
```

and using the **DYNAMIC** scheduling type and chunk size of 100, the chunk sizes would be as follows:

```
100 100 100 100 100 100 100 100 100 100
```

The iterations would be divided into the following chunks:

```
chunk  1 = iterations   1 to  100
chunk  2 = iterations 101 to  200
chunk  3 = iterations 201 to  300
chunk  4 = iterations 301 to  400
chunk  5 = iterations 401 to  500
chunk  6 = iterations 501 to  600
chunk  7 = iterations 601 to  700
chunk  8 = iterations 701 to  800
chunk  9 = iterations 801 to  900
chunk 10 = iterations 901 to 1000
```

A possible scenario for the division of work could be:

```
thread 1 executes chunks 1  5  9
thread 2 executes chunks 2  8
thread 3 executes chunks 3  6  10
thread 4 executes chunks 4  7
```

Example 4. Given the following information:

```
number of iterations = 100
number of threads = 4
```

and using the **STATIC** scheduling type, the iterations would be divided into the following chunks:

```
chunk 1 = iterations  1 to  25
chunk 2 = iterations 26 to  50
chunk 3 = iterations 51 to  75
chunk 4 = iterations 76 to 100
```

A possible scenario for the division of work could be:

```
thread 1 executes chunks 1
thread 2 executes chunks 2
thread 3 executes chunks 3
thread 4 executes chunks 4
```

Related Information

- ″-qhot Option″ in the *User's Guide*
- ″-qsmp Option″ in the *User's Guide*
- “DO” on page 284

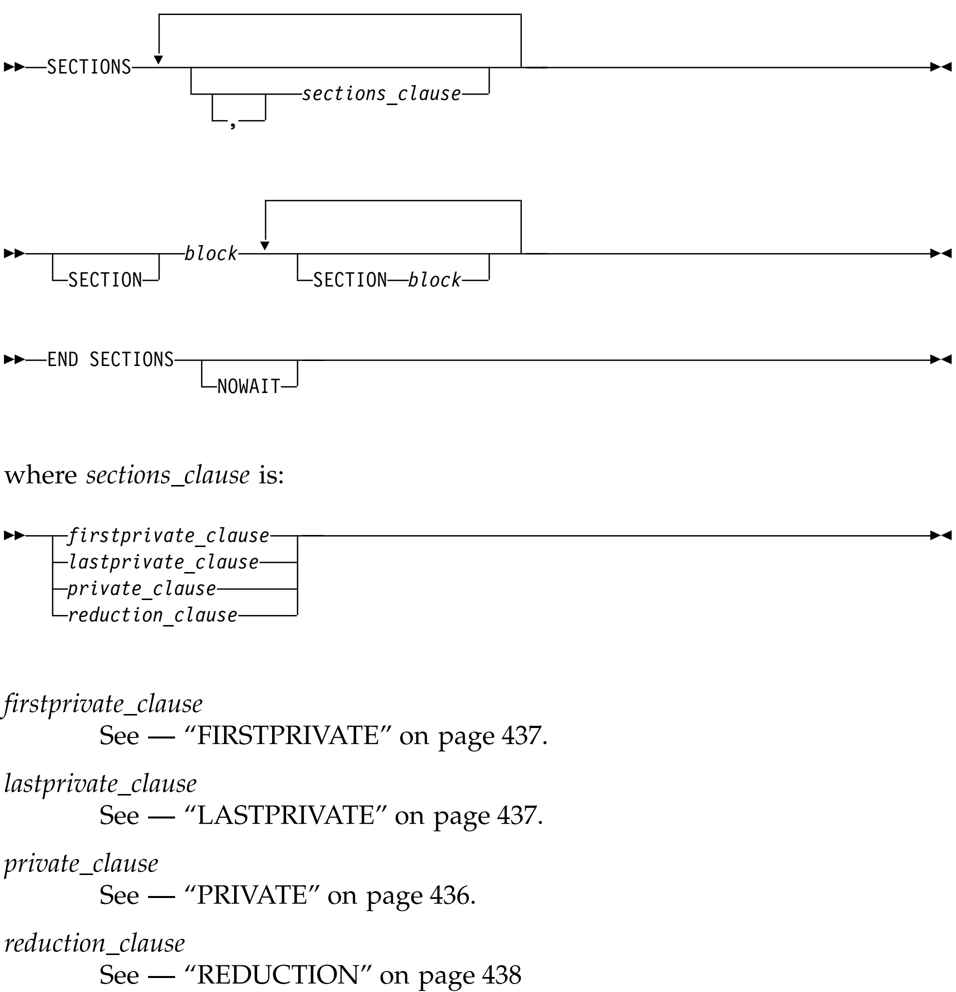
SECTIONS / END SECTIONS

Purpose

The **SECTIONS** construct defines distinct blocks of code to be executed in parallel by threads in the team.

The **SECTIONS** and **END SECTIONS** directives only take effect if you specify the **-qsmp** compiler option.

Format



where *sections_clause* is:

firstprivate_clause
: See — "FIRSTPRIVATE" on page 437.

lastprivate_clause
: See — "LASTPRIVATE" on page 437.

private_clause
: See — "PRIVATE" on page 436.

reduction_clause
: See — "REDUCTION" on page 438

Rules

The **SECTIONS** construct must be encountered by all threads in a team or by none of the threads in a team. All work-sharing constructs and **BARRIER** directives that are encountered must be encountered in the same order by all threads in the team.

The **SECTIONS** construct includes the delimiting directives, and the blocks of code they enclose. At least one block of code must appear in the construct.

You must specify the **SECTION** directive at the beginning of each block of code except for the first. The end of a block is delimited by either another **SECTION** directive or by the **END SECTIONS** directive.

It is illegal to branch into or out of any block of code that is enclosed in the **SECTIONS** construct. All **SECTION** directives must appear within the lexical extent of the **SECTIONS/END SECTIONS** directive pair.

The compiler determines how to divide the work among the threads based on a number of factors, such as the number of threads in the team and the number of sections to be executed in parallel. Therefore, a single thread might execute more than one **SECTION**. It is also possible that a thread in the team might not execute any **SECTION**.

In order for the directive to execute in parallel, you must place the **SECTIONS/END SECTIONS** pair within the dynamic extent of a parallel region. Otherwise, the blocks will be executed serially.

If you specify **NOWAIT** on the **SECTIONS** directive, a thread that completes its sections early will proceed to the instructions following the **SECTIONS** construct. If you do not specify the **NOWAIT** clause, each thread will wait for all of the other threads in the same team to reach the **END SECTIONS** directive. However, there is no implied **BARRIER** at the start of the **SECTIONS** construct.

All work-sharing constructs and **BARRIER** directives that are encountered must be encountered in the same order by all threads in the team.

You cannot specify a **SECTIONS** directive within the dynamic extent of a **CRITICAL** or **MASTER** directive.

You cannot nest **SECTIONS**, **DO** or **SINGLE** directives that bind to the same **PARALLEL** directive.

BARRIER and **MASTER** directives are not permitted in the dynamic extent of a **SECTIONS** directive.

The **END SECTIONS** directive implies the **FLUSH** directive.

Examples

Example 1: This example shows a valid use of the **SECTIONS** construct within a **PARALLEL** region.

```
      INTEGER :: I, B(500), S, SUM
! ...
      S = 0
      SUM = 0
!$OMP PARALLEL SHARED(SUM), FIRSTPRIVATE(S)
!$OMP SECTIONS REDUCTION(+: SUM), LASTPRIVATE(I)
!$OMP SECTION
        S = FCT1(B(1::2))  ! Array B is not altered in FCT1.
        SUM = SUM + S
! ...
!$OMP SECTION
        S = FCT2(B(2::2))  ! Array B is not altered in FCT2.
        SUM = SUM + S
! ...
!$OMP SECTION
        DO I = 1, 500      ! The local copy of S is initialized
          S = S + B(I)     ! to zero.
        END DO
        SUM = SUM + S
! ...
!$OMP END SECTIONS
! ...
!$OMP DO REDUCTION(-: SUM)
      DO J=I-1, 1, -1    ! The loop starts at 500 -- the last
                         ! value from the previous loop.
        SUM = SUM - B(J)
      END DO

!$OMP MASTER
      SUM = SUM - FCT1(B(1::2)) - FCT2(B(2::2))
!$OMP END MASTER
!$OMP END PARALLEL
! ...
                    ! Upon termination of the PARALLEL
                    ! region, the value of SUM remains zero.
```

Example 2: This example shows a valid use of nested **SECTIONS**.

```
!$OMP PARALLEL
!$OMP MASTER
      CALL RANDOM_NUMBER(CX)
      CALL RANDOM_NUMBER(CY)
      CALL RANDOM_NUMBER(CZ)
!$OMP END MASTER

!$OMP SECTIONS
!$OMP SECTION
!$OMP   PARALLEL
!$OMP   SECTIONS PRIVATE(I)
```

```
!$OMP   SECTION
          DO I=1, 5000
            X(I) = X(I) + CX
          END DO
!$OMP   SECTION
          DO I=1, 5000
            Y(I) = Y(I) + CY
          END DO
!$OMP   END SECTIONS
!$OMP   END PARALLEL

!$OMP SECTION
!$OMP   PARALLEL SHARED(CZ,Z)
!$OMP   DO
        DO I=1, 5000
          Z(I) = Z(I) + CZ
        END DO
!$OMP   END DO
!$OMP   END PARALLEL
!$OMP END SECTIONS NOWAIT

! The following computations do not
! depend on the results from the
! previous section.

!$OMP DO
      DO I=1, 5000
        T(I) = T(I) * CT
      END DO
!$OMP END DO
!$OMP END PARALLEL
```

Example 3: This example is invalid. Two nested **SECTIONS** directives cannot bind to the same **PARALLEL** directive.

```
!$OMP PARALLEL SHARED(T)
!$OMP SECTIONS PRIVATE(S)
        X(2::2) = ARRFCT(T)
!$OMP SECTION

!$OMP   SECTIONS REDUCTION(*: S)
!$OMP   SECTION
          S = S*SUM(Y(1::2))
!$OMP   SECTION
          S = S*SUM(Y(2::2))
!$OMP   END SECTIONS

        X(1::2) = ARRFCT(S)
!$OMP END SECTIONS
!$OMP END PARALLEL
```

Related Information

- “PARALLEL / END PARALLEL” on page 476
- “BARRIER” on page 450
- “PARALLEL DO / END PARALLEL DO” on page 479

- “INDEPENDENT” on page 466
- “THREADPRIVATE” on page 510
- ″-qdirective Option″ in the *User's Guide*
- ″-qsmp Option″ in the *User's Guide*

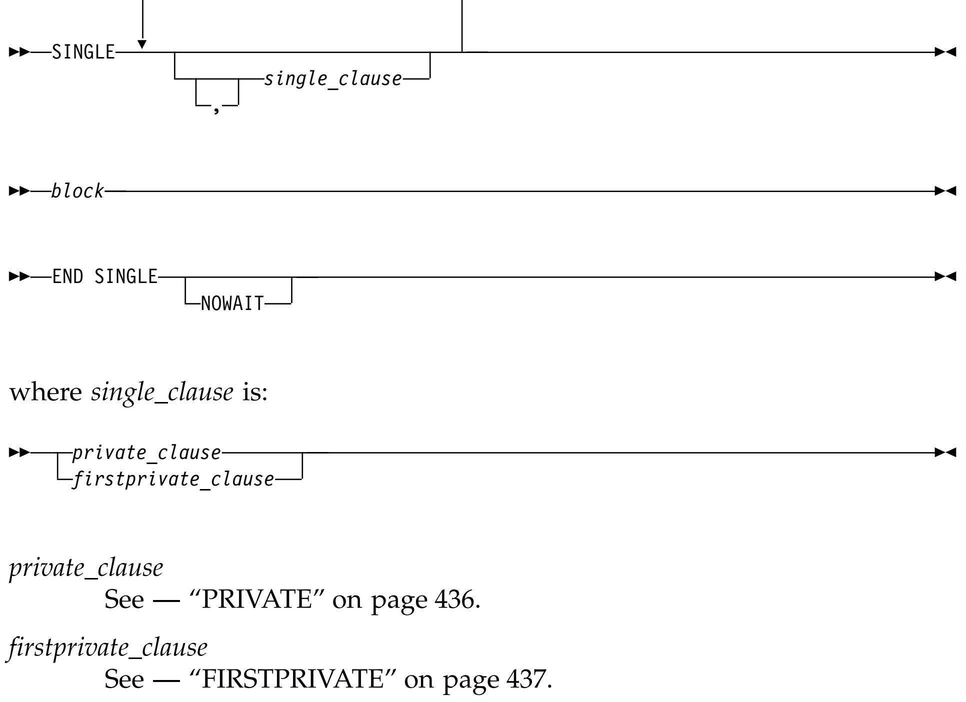
SINGLE / END SINGLE

Purpose

You can use the **SINGLE / END SINGLE** directive construct to specify that the enclosed code should only be executed by one thread in the team.

The **SINGLE** directive only takes effect if you specify the **–qsmp** compiler option.

Format



where *single_clause* is:

private_clause
: See — “PRIVATE” on page 436.

firstprivate_clause
: See — “FIRSTPRIVATE” on page 437.

Rules

It is illegal to branch into or out of a block that is enclosed within the **SINGLE** construct.

The **SINGLE** construct must be encountered by all threads in a team or by none of the threads in a team. All work-sharing constructs and **BARRIER** directives that are encountered must be encountered in the same order by all threads in the team.

If you specify **NOWAIT** on the **SINGLE** directive, the threads that are not executing the **SINGLE** construct will proceed to the instructions following the **SINGLE** construct. If you do not specify the **NOWAIT** clause, each thread will wait at the **END SINGLE** directive until the thread executing the construct reaches the **END SINGLE** directive.

There is no implied **BARRIER** at the start of the **SECTIONS** construct. If you do not specify the **NOWAIT** clause, the **FLUSH** directive is implied at the **END SINGLE** directive.

You cannot nest **SECTIONS**, **DO** and **SINGLE** directives inside one another if they bind to the same **PARALLEL** directive.

SINGLE directives are not permitted within the dynamic extent of **CRITICAL** and **MASTER** directives. **BARRIER** and **MASTER** directives are not permitted within the dynamic extent of **SINGLE** directives.

If you have specified a variable as **PRIVATE**, **FIRSTPRIVATE**, **LASTPRIVATE** or **REDUCTION** in the **PARALLEL** construct which encloses your **SINGLE** construct, you cannot specify the same variable in the **PRIVATE** or **FIRSTPRIVATE** clause of the **SINGLE** construct.

The **SINGLE** directive binds to the closest dynamically enclosing **PARALLEL** directive, if one exists.

Examples

Example 1: In this example, the **BARRIER** directive is used to ensure that all threads finish their work before entering the **SINGLE** construct.

```
      REAL :: X(100), Y(50)
!     ...
!$OMP PARALLEL DEFAULT(SHARED)
      CALL WORK(X)

!$OMP BARRIER
!$OMP SINGLE
      CALL OUTPUT(X)
      CALL INPUT(Y)
!$OMP END SINGLE

      CALL WORK(Y)
!$OMP END PARALLEL
```

Example 2: In this example, the **SINGLE** construct ensures that only one thread is executing a block of code. In this case, array *B* is initialized in the **DO** (work-sharing) construct. After the initialization, a single thread is employed to perform the summation.

```
      INTEGER :: I, J
      REAL :: B(500,500), SM
!     ...

      J = ...
      SM = 0.0
!$OMP PARALLEL
!$OMP DO PRIVATE(I)
      DO I=1, 500
        CALL INITARR(B(I,:), I)       ! initialize the array B
      ENDDO
!$OMP END DO

!$OMP SINGLE                          ! employ only one thread
      DO I=1, 500
        SM = SM + SUM(B(J:J+1,I))
      ENDDO
!$OMP END SINGLE

!$OMP DO PRIVATE(I)
      DO I=500, 1, -1
        CALL INITARR(B(I,:), 501-I)   ! re-initialize the array B
      ENDDO
!$OMP END PARALLEL
```

Example 3: This example shows a valid use of the **PRIVATE** clause. Array *X* is **PRIVATE** to the **SINGLE** construct. If you were to reference array *X* outside of the construct, it would be undefined.

```
      REAL :: X(2000), A(1000), B(1000)

!$OMP PARALLEL
!     ...
!$OMP SINGLE PRIVATE(X)
      CALL READ_IN_DATA(X)
      A = X(1::2)
      B = X(2::2)
!$OMP END SINGLE
!     ...
!$OMP END PARALLEL
```

Example 4: In this example, the **LASTPRIVATE** variable *I* is used in allocating *TMP*, the **PRIVATE** variable in the **SINGLE** construct.

```
      SUBROUTINE ADD(A, UPPERBOUND)
        INTEGER :: A(UPPERBOUND), I, UPPERBOUND
        INTEGER, ALLOCATABLE :: TMP(:)
!  ...
!$OMP   PARALLEL
!$OMP   DO LASTPRIVATE(I)
        DO I=1, UPPERBOUND
          A(I) = I + 1
        ENDDO
!$OMP   END DO
```

```
!$OMP   SINGLE FIRSTPRIVATE(I), PRIVATE(TMP)
        ALLOCATE(TMP(0:I-1))
        TMP = (/ (A(J),J=I,1,-1) /)
!  ...
        DEALLOCATE(TMP)
!$OMP   END SINGLE
!$OMP   END PARALLEL
!  ...
      END SUBROUTINE ADD
```

Example 5: This example is invalid because you cannot nest **SINGLE** directives if they bind to the same **PARALLEL** directive.

```
      INTEGER :: I, A(1000), B(1000), C(1000), T
!  ...
!$OMP PARALLEL
!$OMP SINGLE
      DO I=1, 1000
!$OMP   SINGLE
        B(I) = C(I)*T
!$OMP   END SINGLE

        A(I) = MIN(A(I), B(I))
      ENDDO
!$OMP END SINGLE
!$OMP END PARALLEL
```

Example 6: This example is invalid because a **SINGLE** construct must not appear inside a **CRITICAL** construct.

```
      COMPLEX :: X, Y, Z

!$OMP CRITICAL
!  ...
!$OMP SINGLE
      CALL EXE_ONLY_ONCE(X,Y,Z)
!$OMP END SINGLE
!  ...
!$OMP END CRITICAL
```

Related Information

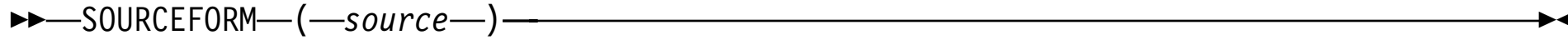
- “BARRIER” on page 450
- “CRITICAL / END CRITICAL” on page 453
- “FLUSH” on page 462
- “MASTER / END MASTER” on page 472
- “PARALLEL / END PARALLEL” on page 476

SOURCEFORM

Purpose

The **SOURCEFORM** compiler directive indicates that all subsequent lines are to be processed in the specified source form until the end of the file is reached or until an **@PROCESS** directive or another **SOURCEFORM** directive specifies a different source form.

Format



source is one of the following: **FIXED**, **FIXED**(*right_margin*), **FREE(F90)**, **FREE(IBM)**, or **FREE**. **FREE** defaults to **FREE(F90)**.

right_margin
: is an unsigned integer specifying the column position of the right margin. The default is 72. The maximum is 132.

Rules

The **SOURCEFORM** directive can appear anywhere within a file. An include file is compiled with the source form of the including file. If the **SOURCEFORM** directive appears in an include file, the source form reverts to that of the including file once processing of the include file is complete.

The **SOURCEFORM** directive cannot specify a label.

Tip

To modify your existing files to Fortran 90 free source form where include files exist:

1. Convert your include files to Fortran 90 free source form: add a **SOURCEFORM** directive to the top of each include file. For example:

   ```
   !CONVERT* SOURCEFORM (FREE(F90))
   ```

   Define your own *trigger_constant* for this conversion process.
2. Once all the include files are converted, convert the .f files. Add the same **SOURCEFORM** directive to the top of each file, or ensure that the .f file is compiled with **-qfree=f90**.
3. Once all files have been converted, you can disable the processing of the directives with the **-qnodirective** compiler option. Ensure that **-qfree=f90** is used at compile time. You may also delete any unnecessary **SOURCEFORM** directives.

Examples

```
@PROCESS DIRECTIVE(CONVERT*)
      PROGRAM MAIN         ! Main program not yet converted
      A=1; B=2
      INCLUDE 'freeform.f'
      PRINT *, RESULT      ! Reverts to fixed form
      END
```

where file freeform.f contains:

```
!CONVERT* SOURCEFORM(FREE(F90))
RESULT = A + B
```

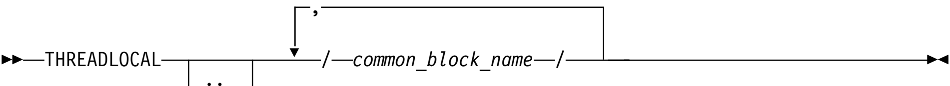
THREADLOCAL

Purpose

You can use the **THREADLOCAL** directive to declare thread-specific common data. It is a possible method of ensuring that access to data that is contained within **COMMON** blocks is serialized.

In order to make use of this directive it is not necessary to specify the **-qsmp** compiler option, but the invocation command must be **xlf_r**, **xlf_r7**, **xlf90_r**, **xlf90_r7**, **xlf95_r**, or **xlf95_r7** to link the necessary libraries.

Format



Rules

You can only declare named blocks as **THREADLOCAL**. All rules and constraints that normally apply to named common blocks apply to common blocks that are declared as **THREADLOCAL**. See "COMMON" on page 267 for more information on the rules and constraints that apply to named common blocks.

The **THREADLOCAL** directive must appear in the *specification_part* of the scoping unit. If a common block appears in a **THREADLOCAL** directive, it must also be declared within a **COMMON** statement in the same scoping unit. The **THREADLOCAL** directive may occur before or after the **COMMON** statement. See "Main Program" on page 152 for more information on the *specification_part* of the scoping unit.

A common block cannot be given the **THREADLOCAL** attribute if it is declared within a **PURE** subprogram.

Members of a **THREADLOCAL** common block must not appear in **NAMELIST** statements.

A common block that is use-associated must not be declared as **THREADLOCAL** in the scoping unit that contains the **USE** statement.

Any pointers declared in a **THREADLOCAL** common block are not affected by the **-qinit=f90ptr** compiler option.

Objects within **THREADLOCAL** common blocks may be used in parallel loops and parallel sections. However, these objects are implicitly shared across the iterations of the loop, and across code blocks within parallel sections. In other words, within a scoping unit, all accessible common blocks, whether declared as **THREADLOCAL** or not, have the **SHARED** attribute within parallel loops and sections in that scoping unit.

If a common block is declared as **THREADLOCAL** within a scoping unit, any subprogram that declares or references the common block, and that is directly or indirectly referenced by the scoping unit, must be executed by the same thread executing the scoping unit. If two procedures that declare common blocks are executed by different threads, then they would obtain different copies of the common block, provided that the common block had been declared **THREADLOCAL**. Threads can be created in one of the following ways:

- Explicitly, via *pthreads* library calls
- Implicitly by the compiler for parallel loop execution
- Implicitly by the compiler for parallel section execution.

If a common block is declared to be **THREADLOCAL** in one scoping unit, it must be declared to be **THREADLOCAL** in every scoping unit that declares the common block.

If a **THREADLOCAL** common block that does not have the **SAVE** attribute is declared within a subprogram, the members of the block become undefined at subprogram RETURN or END, unless there is at least one other scoping unit in which the common block is accessible that is making a direct or indirect reference to the subprogram.

You cannot specify the same *common_block_name* for both a **THREADLOCAL** directive and a **THREADPRIVATE** directive.

Example 1: The following procedure "FORT_SUB" is invoked by two threads:

```
SUBROUTINE FORT_SUB(IARG)
  INTEGER IARG

  CALL LIBRARY_ROUTINE1()
```

```
  CALL LIBRARY_ROUTINE2()
  ...
END SUBROUTINE FORT_SUB

SUBROUTINE LIBRARY_ROUTINE1()
  COMMON /BLOCK/ R              ! The SAVE attribute is required for the common
  SAVE /BLOCK/                  ! block because the program requires that the block
  !IBM* THREADLOCAL /BLOCK/     ! remain defined after library_routine1 is invoked.

  R = 1.0
    ...
END SUBROUTINE LIBRARY_ROUTINE1

SUBROUTINE LIBRARY_ROUTINE2()
  COMMON /BLOCK/ R
  SAVE /BLOCK/
  !IBM* THREADLOCAL /BLOCK/

  ... = R
  ...
END SUBROUTINE LIBRARY_ROUTINE2
```

Example 2: ″FORT_SUB″ is invoked by multiple threads. This is an invalid example because ″FORT_SUB″ and ″ANOTHER_SUB″ both declare /BLOCK/ to be THREADLOCAL. They intend to share the common block, but they are executed by different threads.

```
SUBROUTINE FORT_SUB()
  COMMON /BLOCK/ J
  INTEGER :: J
  !IBM* THREADLOCAL /BLOCK/       ! Each thread executing FORT_SUB
                                 ! obtains its own copy of /BLOCK/
  INTEGER A(10)

  ...
  !IBM* INDEPENDENT
  DO INDEX = 1,10
    CALL ANOTHER_SUB(A(I))
  END DO
  ...

END SUBROUTINE FORT_SUB

SUBROUTINE ANOTHER_SUB(AA)       ! Multiple threads are used to execute ANOTHER_SUB
  INTEGER AA
  COMMON /BLOCK/ J               ! Each thread obtains a new copy of the
  INTEGER :: J                   !  common block /BLOCK/
  !IBM* THREADLOCAL /BLOCK/
  ...
  AA = J                         ! The value of 'J' is undefined.
END SUBROUTINE ANOTHER_SUB
```

Related Information

- ″-qdirective Option″ in the *User's Guide*
- ″-qinit Option″ in the *User's Guide*
- "COMMON" on page 267
- "Main Program" on page 152
- One or more sample programs under the directory **/usr/lpp/xlf/samples/threadlocal** illustrate how to use threadlocal and create threads in C.

THREADPRIVATE

Purpose

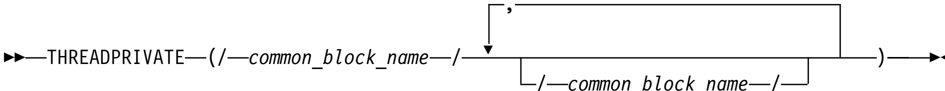
The **THREADPRIVATE** directive ensures that named common blocks are private to a thread but global within the thread. The **THREADPRIVATE** directive ensures that each thread has its own copy of a specific named common block. As a result, the only thread that can directly access the data in a specific named common block is the thread that wrote the data to that common block.

In the serial and **MASTER** sections of a program, only the master thread's copy of the named common block is accessible.

Use the **COPYIN** clause on the **PARALLEL**, **PARALLEL DO**, or **PARALLEL SECTIONS** directive to specify that data in the master thread's copy of the common block should be copied to each thread's private copy of the common block at the beginning of the parallel region.

The **THREADPRIVATE** directive only takes effect if you specify the **-qsmp** compiler option.

Format



common_block_name
is the name of the common block to be made private to a thread.

Rules

You cannot specify a **THREADPRIVATE** common block, or the variables that comprise that block, in a **PRIVATE**, **FIRSTPRIVATE**, **LASTPRIVATE**, **SHARED**, or **REDUCTION** clause.

In **THREADPRIVATE** directives, you can only specify named common blocks.

The **THREADPRIVATE** directive must appear in the *specification_part* of the scoping unit that follows all declarations of the listed common blocks.

You cannot specify the same *common_block_name* for both a **THREADPRIVATE** directive and a **THREADLOCAL** directive.

All rules and constraints that apply to named common blocks also apply to common blocks declared as **THREADPRIVATE**. See "COMMON" on page 267.

If you declare a common block as **THREADPRIVATE** in one scoping unit, you must declare it as **THREADPRIVATE** in all other scoping units in which it is declared.

If a **THREADPRIVATE** common block is not initialized or you do not assign a value to it before the start of the first parallel region, that common block is undefined. When a common block that has been initialized appears in a **THREADPRIVATE** directive, each thread's copy of the block is initialized before it is used for the first time. If you assigned a value to a **THREADPRIVATE** common block before the start of the first parallel region and you do not specify a **COPYIN** clause, the value is only assigned to the master thread's copy of the common block. If, however, you do specify a **COPYIN** clause, the value is copied from the master thread's copy of the common block to each thread's copy of the common block (or to one of its constituents).

For the second and subsequent parallel regions in a program, the data in a **THREADPRIVATE** common block persists only if the dynamic threads mechanism has been disabled and if the number of threads is the same for all of the parallel regions.

You cannot access the name of a common block by use association or host association. Thus, a named common block can only appear on a **THREADPRIVATE** directive if the common block is declared in the scoping unit that contains the **THREADPRIVATE** directive. However, you can access the variables in the common block by use association or host association. For more information, see "Host Association" on page 137 and "Use Association" on page 139.

The **-qinit=f90ptr** compiler option does not affect pointers that you have declared in a **THREADPRIVATE** common block.

The **DEFAULT** clause does not affect variables in **THREADPRIVATE** common blocks.

Examples

Example 1: In this example, the **PARALLEL DO** directive invokes multiple threads that call **SUB1**. The common block **BLK** in **SUB1** shares the data that is specific to the thread with subroutine **SUB2**, which is called by **SUB1**.

```
      PROGRAM TT
        INTEGER :: I, B(50)

!$OMP   PARALLEL DO SCHEDULE(STATIC, 10)
        DO I=1, 50
          CALL SUB1(I, B(I))       ! Multiple threads call SUB1.
        ENDDO
      END PROGRAM TT

      SUBROUTINE SUB1(J, X)
        INTEGER :: J, X, A(100)
        COMMON /BLK/ A
!$OMP   THREADPRIVATE(/BLK/)   ! Array a is private to each thread.
!  ...
        CALL SUB2(J)
        X = A(J) + A(J + 50)
!  ...
      END SUBROUTINE SUB1

      SUBROUTINE SUB2(K)
        INTEGER :: C(100)
        COMMON /BLK/ C
!$OMP   THREADPRIVATE(/BLK/)
!  ...
        C = K
!  ...                         ! Since each thread has its own copy of
                               ! common block BLK, the assignment of
                               ! array C has no effect on the copies of
                               ! that block owned by other threads.
END SUBROUTINE SUB2
```

Example 2: In this example, each thread has its own copy of the common block **ARR** in the parallel section. If one thread initializes the common block variable **TEMP**, the initial value is not visible to other threads.

```
      PROGRAM ABC
        INTEGER :: I, TEMP(100), ARR1(50), ARR2(50)
        COMMON /ARR/ TEMP
!$OMP   THREADPRIVATE(/ARR/)
        INTERFACE
          SUBROUTINE SUBS(X)
            INTEGER :: X(:)
          END SUBROUTINE
        END INTERFACE
! ...
!$OMP   PARALLEL SECTIONS
!$OMP   SECTION                     ! The thread has its own copy of the
! ...                               ! common block ARR.
          TEMP(1:100:2) = -1
          TEMP(2:100:2) = 2
```

```
          CALL SUBS(ARR1)
! ...
!$OMP   SECTION                   ! The thread has its own copy of the
! ...                             ! common block ARR.
          TEMP(1:100:2) = 1
          TEMP(2:100:2) = -2
          CALL SUBS(ARR2)
! ...
!$OMP    END PARALLEL SECTIONS
! ...
        PRINT *, SUM(ARR1), SUM(ARR2)
      END PROGRAM ABC

      SUBROUTINE SUBS(X)
        INTEGER :: K, X(:), TEMP(100)
        COMMON /ARR/ TEMP
!$OMP   THREADPRIVATE(/ARR/)
!  ...
        DO K = 1, UBOUND(X, 1), 2
          X(K) = TEMP(K) + TEMP(K + 1)    ! The thread is accessing its
                                          ! own copy of
                                          ! the common block.
        ENDDO
! ...
      END SUBROUTINE SUBS
```

The expected output for this program is:

```
50 -50
```

Example 3: This example is invalid because constituents of the **THREADPRIVATE** common block cannot appear in a **SHARED** clause.

```
      INTEGER :: I
      COMMON /CB1/ A(500)
!$OMP THREADPRIVATE(/CB1/)

!$OMP PARALLEL DO SHARED(A)
      DO I=1, 500
        CALL SUB(A(I))
      ENDDO
! ...
```

Example 4: In this invalid example, variables in a common block are undefined in a parallel region because they do not appear in a **COPYIN** clause.

```
      REAL :: X, Y, Z
      COMMON /COORD/ X, Y, Z
!$OMP THREADPRIVATE(/COORD/)
!$OMP PARALLEL
!  ...
      DIST = X*X + Y*Y + Z*Z  ! x, y, and z are undefined in the parallel
```

```
                                     ! region because they do not appear in a
!  ...                               ! COPYIN clause.
!$OMP END PARALLEL
!  ...
```

Related Information

- "COMMON" on page 267
- **OMP_DYNAMIC** environment variable in the *User's Guide*.
- "omp_set_dynamic" on page 673
- "PARALLEL / END PARALLEL" on page 476
- "PARALLEL DO / END PARALLEL DO" on page 479
- "PARALLEL SECTIONS / END PARALLEL SECTIONS" on page 483

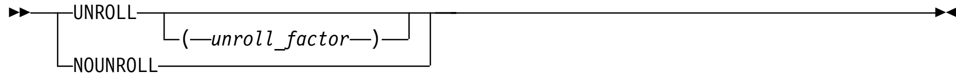
UNROLL

Purpose

Loop unrolling consists of replicating the body of a loop in order to reduce the number of iterations required to complete the loop. The **UNROLL** directive indicates to the compiler that the **DO** loop that immediately follows the directive can be unrolled.

The **UNROLL**, **NOUNROLL** and **UNROLL**(*n*) directives take precedence over the **-qunroll** and **-qnounroll** compiler options.

Format



unroll_factor
: must be a positive scalar integer initialization expression. The value of *unroll_factor* must be greater than 1 if you want unrolling to occur.

Rules

The **UNROLL** directive permits the compiler to perform unrolling on the specified loop, but the compiler is not required to perform this activity.

Unrolling is only allowed on the innermost **DO** loop, that is, a loop that contains no other **DO** loops. You cannot unroll loops that are introduced by implied-**DO** loops or the use of Fortran array language. Also, you cannot specify the **UNROLL** or **UNROLL**(*unroll_factor*) directives for **DO WHILE** loops or infinite **DO** loops.

The **UNROLL** and **UNROLL**(*unroll_factor*) directives should immediately precede an innermost **DO** loop.

The **UNROLL**(*unroll_factor*) directive requests that the compiler perform unrolling on the innermost loop *unroll_factor* times, if *unroll_factor* is greater than 1. If *unroll_factor* is 1, unrolling is disabled.

Specifying **NOUNROLL** is equivalent to specifying **UNROLL**(1).

You cannot specify more than one **UNROLL**, **UNROLL** (*unroll_factor*), or **NOUNROLL** directive on the same **DO** loop.

Examples

Example 1: In this example, the **UNROLL(2)** directive is used to tell the compiler that the body of the loop can be replicated so that the work of two iterations is performed in a single iteration. Instead of performing 1000 iterations, if the compiler unrolls the loop, it will only perform 500 iterations.

```
!IBM* UNROLL(2)
      DO I = 1, 1000
         A(I) = I
      END DO
```

If the compiler chooses to unroll the previous loop, the compiler translates the loop so that it is essentially equivalent to the following:

```
     DO I = 1, 1000, 2
        A(I) = I
        A(I+1) = I + 1
     END DO
```

Example 2: In the first **DO** loop, **UNROLL(3)** is used. If unrolling is performed, the compiler will unroll the loop so that the work of three iterations is done in a single iteration. In the second **DO** loop, the compiler determines how to unroll the loop for maximum performance.

```
      PROGRAM GOODUNROLL

      INTEGER I, X(1000)
      REAL A, B, C, TEMP, Y(1000)

!IBM* UNROLL(3)
      DO I = 1, 1000
         X(I) = X(I) + 1
      END DO

!IBM* UNROLL
      DO I = 1, 1000
         A = -I
         B = I + 1
         C = I + 2
         TEMP = SQRT(B*B - 4*A*C)
         Y(I) = (-B + TEMP) / (2*A)
      END DO
      END PROGRAM GOODUNROLL
```

Related Information

- "DO Construct" on page 126.

- ″-qsmp Option″ in the *User's Guide*
- ″-qunroll Option″ in the *User's Guide*

Part 4. Intrinsic and Library Procedures

Part 4 provides information on:

- “Chapter 12. Intrinsic Procedures” on page 519
- “Chapter 13. Service and Utility Procedures” on page 639
- “Chapter 14. OpenMP Execution Environment Routines and Lock Routines” on page 667
- “Chapter 15. Pthreads Library Module” on page 677

© Copyright IBM Corp. 1990, 2000

Chapter 12. Intrinsic Procedures

Fortran defines a number of procedures, called intrinsic procedures, that are available to any program. This chapter provides an alphabetical reference to these procedures.

Related Information:

1. "Intrinsic Procedures" on page 160 provides background information that you may need to be familiar with before proceeding with this chapter.
2. "INTRINSIC" on page 345 is a related statement.

Classes of Intrinsic Procedures

There are five classes of intrinsic procedures: inquiry functions, elemental procedures, system inquiry functions, transformational functions, and subroutines.

Inquiry Intrinsic Functions

The result of an *inquiry function* depends on the properties of its principal argument, not on the value of the argument. The value of the argument does not have to be defined.

ALLOCATED	LEN	RADIX
ASSOCIATED	LOC	RANGE
BIT_SIZE	MAXEXPONENT	SHAPE
DIGITS	MINEXPONENT	SIZE
EPSILON	NUM_PARTHDS	TINY
HUGE	NUM_USRTHDS	UBOUND
KIND	PRECISION	
LBOUND	PRESENT	

Elemental Intrinsic Procedures

Some intrinsic functions and one intrinsic subroutine (**MVBITS**) are *elemental*. That is, they can be specified for scalar arguments, but also accept arguments that are arrays.

If all arguments are scalar, the result is a scalar.

If any argument is an array, all INTENT(OUT) and INTENT(INOUT) arguments must be arrays of the same shape, and the remaining arguments must be conformable with them.

© Copyright IBM Corp. 1990, 2000

The shape of the result is the shape of the argument with the greatest rank. The elements of the result are the same as if the function was applied individually to the corresponding elements of each argument.

ABS	EXPONENT	LOG
ACHAR	FLOOR	LOG10
ACOS	FMADD	LOGICAL
ACOSD	FMSUB	LSHIFT
ADJUSTL	FNMADD	MAX
ADJUSTR	FNMSUB	MERGE
AIMAG	FRE	MIN
AINT	FRSQRTE	MOD
ANINT	FSEL	MODULO
ASIN	FRACTION	MVBITS
ASIND	GAMMA	NEAREST
ATAN	HFIX	NINT
ATAND	IACHAR	NOT
ATAN2	IBCLR	QCMPLX
ATAN2D	IBITS	QEXT
BTEST	IBSET	REAL
CEILING	ICHAR	RRSPACING
CHAR	IEOR	RSHIFT
CMPLX	ILEN	SCALE
CONJG	INDEX	SCAN
COS	INT	SET_EXPONENT
COSD	IOR	SIGN
COSH	ISHFT	SIN
CVMGx	ISHFTC	SIND
DBLE	LEADZ	SINH
DCMPLX	LEN_TRIM	SPACING
DIM	LGAMMA	SQRT
DPROD	LGE	TAN
ERF	LGT	TAND
ERFC	LLE	TANH
EXP	LLT	VERIFY

System Inquiry Intrinsic Functions

The *system inquiry functions* may be used in restricted expressions. They cannot be used in initialization expressions, nor can they be passed as actual arguments.

NUMBER_OF_PROCESSORS
PROCESSORS_SHAPE

Transformational Intrinsic Functions

All other intrinsic functions are classified as *transformational functions*. They generally accept array arguments and return array results that depend on the values of elements in the argument arrays.

ALL	MAXVAL	SELECTED_INT_KIND
ANY	MINLOC	SELECTED_REAL_KIND
COUNT	MINVAL	SPREAD
CSHIFT	NULL	SUM
DOT_PRODUCT	PACK	TRANSFER
EOSHIFT	PRODUCT	TRANSPOSE
MATMUL	REPEAT	TRIM
MAXLOC	RESHAPE	UNPACK

For background information on arrays, see "Chapter 4. Array Concepts" on page 63.

Intrinsic Subroutines

Some intrinsic procedures are subroutines. They perform a variety of tasks.

ABORT	MVBITS	SRAND
CPU_TIME	RANDOM_NUMBER	SYSTEM
DATE_AND_TIME	RANDOM_SEED	SYSTEM_CLOCK
GETENV	SIGNAL	

Data Representation Models

Integer Bit Model

The following model shows how the processor represents each bit of a nonnegative scalar integer object:

$$j = \sum_{k=0}^{s-1} w_k \times 2^k$$

j is the integer value

s is the number of bits

w_k is binary digit w located at position k

XL Fortran implements the following s parameters for the XL Fortran integer kind type parameters:

Integer Kind Parameter	s Parameter
1	8
2	16
4	32
8	64

The following intrinsic functions use this model:

BTEST	IBSET	ISHFTC
IAND	IEOR	MVBITS
IBCLR	IOR	NOT
IBITS	ISHFT	

Integer Data Model

$$i = s \times \sum_{k=1}^{q} w_k \times r^{k-1}$$

i is the integer value

s is the sign (±1)

q is the number of digits (positive integer)

w_k is a nonnegative digit < *r*

r is the radix

XL Fortran implements this model with the following *r* and *q* parameters:

Integer Kind Parameter	*r* Parameter	*q* Parameter
1	2	7
2	2	15
4	2	31
8	2	63

The following intrinsic functions use this model:

DIGITS	RADIX	RANGE
HUGE		

Real Data Model

$$x = \begin{cases} 0 & \text{or} \\ s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k} & \end{cases}$$

x is the real value

s is the sign (±1)

b is an integer > 1

e is an integer, where $e_{min} \le e \le e_{max}$

p is an integer > 1

f_k is a nonnegative integer < b ($f_1 \ne 0$)

XL Fortran implements this model with the following parameters:

Real Kind parameter	b Parameter	p Parameter	e_{min} Parameter	e_{max} Parameter
4	2	24	-125	128
8	2	53	-1021	1024
16	2	106	-1021	1024

The following intrinsic functions use this model:

DIGITS	MINEXPONENT	RRSPACING
EPSILON	NEAREST	SCALE
EXPONENT	PRECISION	SET_EXPONENT
FRACTION	RADIX	SPACING
HUGE	RANGE	TINY
MAXEXPONENT		

Detailed Descriptions of Intrinsic Procedures

The following is an alphabetical list of all generic names for intrinsic procedures.

For each procedure, several items of information are listed.

Notes:

1. The argument names listed in the title can be used as the names for keyword arguments when calling the procedure.
2. For those procedures with specific names, a table lists each specific name along with information about the specific function:
   - When a function return type or argument type is shown in lowercase, that indicates that the type is specified as shown, but the compiler may actually substitute a call to a different specific name depending on the settings of the **-qintsize**, **-qrealsize**, and **-qautodbl** options.

     For example, references to **SINH** are replaced by references to **DSINH** when **-qrealsize=8** is in effect, and references to **DSINH** are replaced by references to **QSINH**.

- The column labeled "Pass as Arg?" indicates whether or not you can pass that specific name as an actual argument to a procedure. Only the specific name of an intrinsic procedure may be passed as an actual argument, and only for some specific names. A specific name passed this way may only be referenced with scalar arguments.

3. The index contains entries for each specific name, if you know the specific name but not the generic one.

ABORT()

Terminates the program. It truncates all open output files to the current position of the file pointer, closes all open files, and sends the SIGIOT signal to the current process.

If the SIGIOT is neither caught nor ignored, and if the current directory is writable, the system produces a core file in the current directory.

Class

Subroutine

Examples

The following is an example of a statement using the ABORT subroutine.

```
IF (ERROR_CONDITION) CALL ABORT
```

The following is the output generated by the above program:

```
 /home/mark
IOT/Abort trap(coredump)
```

ABS(A)

Absolute value.

A must be of type integer, real, or complex.

Class

Elemental function

Result Type and Attributes

The same as A, except that if A is complex, the result is real.

Result Value

- If A is of type integer or real, the result is $|A|$.
- If A is of type complex with value (x,y), the result approximates

$$\sqrt{x^2 + y^2}$$

Examples

ABS ((3.0, 4.0)) has the value 5.0.

Specific Name	Argument Type	Result Type	Pass As Arg?
IABS	any integer 1	same as argument	yes
ABS	default real	default real	yes
DABS	double precision real	double precision real	yes
QABS	REAL(16)	REAL(16)	yes
CABS	default complex	default real	yes
CDABS	double complex	double precision real	yes
ZABS	double complex	double precision real	yes
CQABS	COMPLEX(16)	REAL(16)	yes

Notes:

1. The extension is the ability to specify a nondefault integer argument.

ACHAR(I)

Returns the character in a specified position of the ASCII collating sequence. It is the inverse of the IACHAR function.

I must be of type integer.

Class

Elemental function

Result Type and Attributes

Character of length one with the same kind type parameter as KIND ('A').

Result Value

- If I has a value in the range $0 \le I \le 127$, the result is the character in position I of the ASCII collating sequence, provided that the character corresponding to I is representable.
- If I is outside the allowed value range, the result is undefined.

Examples

ACHAR (88) has the value 'X'.

ACOS(X)

Arccosine (inverse cosine) function.

X must be of type real with a value that satisfies the inequality $|X| \le 1$.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

- It is expressed in radians, and approximates arccos(X).
- It is in the range $0 \leq \text{ACOS(X)} \leq \pi$.

Examples

ACOS (1.0) has the value 0.0.

Specific Name	Argument Type	Result Type	Pass As Arg?
ACOS	default real	default real	yes
DACOS	double precision real	double precision real	yes
QACOS	REAL(16)	REAL(16)	yes
QARCOS	REAL(16)	REAL(16)	yes

ACOSD(X)

Arccosine (inverse cosine) function. Result in degrees.

X must be of type real. Its value must satisfy the inequality $|X| \leq 1$.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

- It is expressed in degrees and approximates arccos(X).
- It is in the range $0° \leq \text{ACOSD(X)} \leq 180°$.

Examples

ACOSD (0.5) has the value 60.0.

Specific Name	Argument Type	Result Type	Pass As Arg?
ACOSD	default real	default real	yes
DACOSD	double precision real	double precision real	yes
QACOSD	REAL(16)	REAL(16)	yes

ADJUSTL(STRING)

Adjust to the left, removing leading blanks and inserting trailing blanks.

STRING must be of type character.

Class

Elemental function

Result Type and Attributes

Character of the same length and kind type parameter as STRING.

Result Value

The value of the result is the same as STRING except that any leading blanks have been deleted and the same number of trailing blanks have been inserted.

Examples

ADJUSTL ('ƀWORD') has the value 'WORDƀ'.

ADJUSTR(STRING)

Adjust to the right, removing trailing blanks and inserting leading blanks.

STRING must be of type character.

Class

Elemental function

Result Type and Attributes

Character of the same length and kind type parameter as STRING.

Result Value

The value of the result is the same as STRING except that any trailing blanks have been deleted and the same number of leading blanks have been inserted.

Examples

ADJUSTR ('WORDƀ') has the value 'ƀWORD'.

AIMAG(Z), IMAG(Z)

Imaginary part of a complex number.

Z must be of type complex.

Class

Elemental function

Result Type and Attributes

Real with the same kind type parameter as Z.

Result Value

If Z has the value (x,y), the result has the value y.

Examples

AIMAG ((2.0, 3.0)) has the value 3.0.

Specific Name	Argument Type	Result Type	Pass As Arg?
AIMAG	default complex	default real	yes
DIMAG	double complex	double precision real	yes
QIMAG	COMPLEX(16)	REAL(16)	yes

AINT(A, KIND)

Truncates to a whole number.

A must be of type real.

KIND (optional)
must be a scalar integer initialization expression.

Class

Elemental function

Result Type and Attributes

- The result type is real.
- If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of A.

Result Value

- If $|A| < 1$, the result is zero.
- If $|A| \geq 1$, the result has a value equal to the integer whose magnitude is the largest integer that does not exceed the magnitude of A and whose sign is the same as the sign of A.

Examples

```
AINT(3.555) = 3.0
AINT(-3.555) = -3.0
```

Specific Name	Argument Type	Result Type	Pass As Arg?
AINT	default real	default real	yes
DINT	double precision real	double precision real	yes
QINT	REAL(16)	REAL(16)	yes

ALL(MASK, DIM)

Determines if all values in an entire array, or in each vector along a single dimension, are true.

MASK
is a logical array.

DIM (optional)
is an integer scalar in the range 1 ≤ DIM ≤ rank(MASK). The corresponding actual argument must not be an optional dummy argument.

Class

Transformational function

Result Value

The result is a logical array with the same type and type parameters as MASK, and rank rank(MASK)-1. If the DIM is missing, or MASK has a rank of one, the result is a scalar of type logical.

The shape of the result is (s_1, s_2, ..., $s_{(DIM-1)}$, $s_{(DIM+1)}$, ..., s_n), where n is the rank of MASK.

Each element in the result array is .TRUE. only if all the elements given by MASK(m_1, m_2, ..., $m_{(DIM-1)}$, :, $m_{(DIM+1)}$, ..., m_n), are true. When the result is a scalar, either because DIM is not specified or because MASK is of rank one, it is .TRUE. only if all elements of MASK are true, or MASK has size zero.

Examples

```
! A is the array | 4 3 6 |, and B is the array | 3 5 2 |
!                | 2 4 1 |                     | 7 8 4 |

! Is every element in A less than the
! corresponding one in B?
      RES = ALL(A .LT. B)          ! result RES is false

! Are all elements in each column of A less than the
! corresponding column of B?
      RES = ALL(A .LT. B, DIM = 1) ! result RES is (f,t,f)

! Same question, but for each row of A and B.
      RES = ALL(A .LT. B, DIM = 2) ! result RES is (f,t)
```

ALLOCATED(ARRAY)

Indicate whether or not an allocatable array is currently allocated.

ARRAY
is an allocatable array whose allocation status you want to know.

Class

Inquiry function

Result Type and Attributes

Default logical scalar.

Result Value

The result corresponds to the allocation status of ARRAY: .TRUE. if it is currently allocated, .FALSE. if it is not currently allocated, or undefined if its allocation status is undefined. If you are compiling with the **-qxlf90=autodealloc** compiler option there is no undefined allocation status.

Examples

```
INTEGER, ALLOCATABLE, DIMENSION(:) :: A
PRINT *, ALLOCATED(A)      ! A is not allocated yet.
ALLOCATE (A(1000))
PRINT *, ALLOCATED(A)      ! A is now allocated.
END
```

Related Information

"Allocatable Arrays" on page 70, "ALLOCATE" on page 246, "Allocation Status" on page 58.

ANINT(A, KIND)

Nearest whole number.

A
must be of type real.

KIND (optional)
must be a scalar integer initialization expression.

Class

Elemental function

Result Type and Attributes

- The result type is real.
- If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of A.

Result Value

- If A $>$ 0, ANINT(A) = AINT(A + 0.5)
- If A $\leq$ 0, ANINT(A) = AINT(A - 0.5)

Note: The addition and subtraction of 0.5 are done in round-to-zero mode.

Examples

```
ANINT(3.555) = 4.0
ANINT(-3.555) = -4.0
```

Specific Name	Argument Type	Result Type	Pass As Arg?
ANINT	default real	default real	yes
DNINT	double precision real	double precision real	yes
QNINT	REAL(16)	REAL(16)	yes

ANY(MASK, DIM)

Determines if any of the values in an entire array, or in each vector along a single dimension, are true.

MASK
is a logical array.

DIM (optional)
is an integer scalar in the range 1 ≤ DIM ≤ rank(MASK). The corresponding actual argument must not be an optional dummy argument.

Class

Transformational function

Result Value

The result is a logical array of the same type and type parameters as MASK, and rank of rank(MASK)-1. If the DIM is missing, or MASK has a rank of one, the result is a scalar of type logical.

The shape of the result is (s_1, s_2, ..., $s_{(DIM\ -1)}$, $s_{(DIM+1)}$, ..., s_n), where n is the rank of MASK.

Each element in the result array is .TRUE. if any of the elements given by MASK(m_1, m_2, ..., $m_{(DIM-1)}$, :, $m_{(DIM+1)}$, ..., m_n) are true. When the result is a scalar, either because DIM is not specified or because MASK is of rank one, it is .TRUE. if any of the elements of MASK are true.

Examples

```
! A is the array | 9 -6 7 |, and B is the array | 2 7 8 |
!                | 3 -1 5 |                     | 5 6 9 |

! Is any element in A greater than or equal to the
! corresponding element in B?
      RES = ANY(A .GE. B)          ! result RES is true

! For each column in A, is there any element in the column
! greater than or equal to the corresponding element in B?
      RES = ANY(A .GE. B, DIM = 1) ! result RES is (t,f,f)

! Same question, but for each row of A and B.
      RES = ANY(A .GE. B, DIM = 2) ! result RES is (t,f)
```

ASIN(X)

Arcsine (inverse sine) function.

X must be of type real. Its value must satisfy the inequality $|X| \leq 1$.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

- It is expressed in radians, and approximates arcsin(X).
- It is in the range $-\pi/2 \leq \text{ASIN}(X) \leq \pi/2$.

Examples

ASIN (1.0) approximates $\pi/2$.

Specific Name	Argument Type	Result Type	Pass As Arg?
ASIN	default real	default real	yes
DASIN	double precision real	double precision real	yes
QASIN	REAL(16)	REAL(16)	yes
QARSIN	REAL(16)	REAL(16)	yes

ASIND(X)

Arcsine (inverse sine) function. Result in degrees.

X must be of type real. Its value must satisfy the inequality $|X| \leq 1$.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

- It is expressed in degrees, and approximates arcsin(X).
- It is in the range $-90° \leq \text{ASIND}(X) \leq 90°$

Examples

ASIND (0.5) has the value 30.0.

Specific Name	Argument Type	Result Type	Pass As Arg?
ASIND	default real	default real	yes

Specific Name	Argument Type	Result Type	Pass As Arg?
DASIND	double precision real	double precision real	yes
QASIND	REAL(16)	REAL(16)	yes

ASSOCIATED(POINTER, TARGET)

Returns the association status of its pointer argument, or indicates whether the pointer is associated with the target.

POINTER A pointer whose association status you want to test. It can be of any type. Its association status must not be undefined.

TARGET (optional)
A pointer or target that might or might not be associated with POINTER. Its association status must not be undefined.

Class

Inquiry function

Result Type and Attributes

Default logical scalar.

Result Value

If only the POINTER argument is specified, the result is .TRUE. if it is associated with any target and .FALSE. otherwise. If TARGET is also specified, the procedure tests whether POINTER is associated with TARGET, or with the same object that TARGET is associated with (if TARGET is also pointer).

The result is undefined if either POINTER or TARGET is associated with a zero-sized array, or if TARGET is a zero-sized array.

Objects with different types or shapes cannot be associated with each other.

Arrays with the same type and shape but different bounds can be associated with each other.

Examples

```
REAL, POINTER, DIMENSION(:,:) :: A
REAL, TARGET, DIMENSION(5,10) :: B, C

NULLIFY (A)
PRINT *, ASSOCIATED (A)   ! False, not associated yet

A => B
PRINT *, ASSOCIATED (A)   ! True, because A is
                          ! associated with B
```

```
PRINT *, ASSOCIATED (A,C) ! False, A is not
                          ! associated with C
END
```

ATAN(X)

Arctangent (inverse tangent) function.

X must be of type real.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

- It is expressed in radians and approximates arctan(X).
- It is in the range $-\pi/2 \leq$ ATAN(X) $\leq \pi/2$.

Examples

ATAN (1.0) approximates $\pi/4$.

Specific Name	Argument Type	Result Type	Pass As Arg?
ATAN	default real	default real	yes
DATAN	double precision real	double precision real	yes
QATAN	REAL(16)	REAL(16)	yes

ATAND(X)

Arctangent (inverse tangent) function. Result in degrees.

X must be of type real.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

- It is expressed in degrees and approximates arctan(X).
- It is in the range $-90° \leq$ ATAND(X) $\leq 90°$.

Examples

ATAND (1.0) has the value 45.0.

Specific Name	Argument Type	Result Type	Pass As Arg?
ATAND	default real	default real	yes
DATAND	double precision real	double precision real	yes
QATAND	REAL(16)	REAL(16)	yes

ATAN2(Y, X)

Arctangent (inverse tangent) function. The result is the principal value of the nonzero complex number (X, Y) formed by the real arguments Y and X.

Y must be of type real.

X must be of the same type and kind type parameter as Y. If Y has the value zero, X must not have the value zero.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

- It is expressed in radians and has a value equal to the principal value of the argument of the complex number (X, Y).
- It is in the range $-\pi < \text{ATAN2}(Y, X) \leq \pi$.
- If $X \neq 0$, the result approximates arctan(Y/X).
- If $Y > 0$, the result is positive.
- If $Y < 0$, the result is negative.
- If $Y = 0$ and $X > 0$, the result is zero.
- If $Y = 0$ and $X < 0$, the result is π.
- If $X = 0$, the absolute value of the result is $\pi/2$.

Examples

ATAN2 (1.5574077, 1.0) has the value 1.0.

Given that:

```
Y = |  1   1 |          X = | -1  1 |
    | -1  -1 |              | -1  1 |
```

the value of ATAN2(Y,X) is approximately:

```
ATAN2 (Y, X) = |  3π/4   π/4 |
               | -3π/4  -π/4 |
```

Specific Name	Argument Type	Result Type	Pass As Arg?
ATAN2	default real	default real	yes

Specific Name	Argument Type	Result Type	Pass As Arg?
DATAN2	double precision real	double precision real	yes
QATAN2	REAL(16)	REAL(16)	yes

ATAN2D(Y, X)

Arctangent (inverse tangent) function. The result is the principal value of the nonzero complex number (X, Y) formed by the real arguments Y and X.

Y must be of type real.

X must be of the same type and kind type parameter as Y. If Y has the value zero, X must not have the value zero.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

- It is expressed in degrees and has a value equal to the principal value of the argument of the complex number (X, Y).
- It is in the range -180° < ATAN2D(Y,X) ≤ 180°.
- If X≠0, the result approximates arctan(Y/X).
- If Y>0, the result is positive.
- If Y<0, the result is negative.
- If Y=0 and X>0, the result is zero.
- If Y=0 and X<0, the result is 180°.
- If X=0, the absolute value of the result is 90°.

Examples

ATAN2D (1.5574077, 1.0) has the value 57.295780181 (approximately).

Given that:

```
Y = |  1.0   1.0 |     X = | -1.0   1.0 |
    | -1.0  -1.0 |         | -1.0   1.0 |
```

then the value of ATAN2D(Y,X) is:

```
ATAN2D(Y,X) =  |  135.0000000   45.00000000 |
               | -135.0000000  -45.00000000 |
```

Specific Name	Argument Type	Result Type	Pass As Arg?
ATAN2D	default real	default real	yes
DATAN2D	double precision real	double precision real	yes

Specific Name	Argument Type	Result Type	Pass As Arg?
QATAN2D	REAL(16)	REAL(16)	yes

BIT_SIZE(I)

Returns the number of bits in an integer type. Because only the type of the argument is examined, the argument need not be defined.

I must be of type integer.

Class

Inquiry function

Result Type and Attributes

Scalar integer with the same kind type parameter as I.

Result Value

The result is the number of bits in the integer data type of the argument:

```
    type                   bits
-----------               ------
 integer(1)                  08
 integer(2)                  16
 integer(4)                  32
 integer(8)                  64
```

The bits are numbered from 0 to BIT_SIZE(I)-1, from right to left.

Examples

BIT_SIZE (1_4) has the value 32, because the integer type with kind 4 (that is, a four-byte integer) contains 32 bits.

BTEST(I, POS)

Tests a bit of an integer value.

I must be of type integer.

POS must be of type integer. It must be nonnegative and be less than BIT_SIZE(I).

Class

Elemental function

Result Type and Attributes

The result is of type default logical.

Result Value

The result has the value .TRUE. if bit POS of I has the value 1 and the value .FALSE. if bit POS of I has the value 0.

The bits are numbered from 0 to BIT_SIZE(I)-1, from right to left.

Examples

BTEST (8, 3) has the value .TRUE..

```
If A has the value
   | 1 2 |
   | 3 4 |
the value of BTEST (A, 2) is
   | false false |
   | false true  |
and the value of BTEST (2, A) is
   | true  false |
   | false false |
```

See "Integer Bit Model" on page 521.

Specific Name	Argument Type	Result Type	Pass As Arg?
BTEST	any integer	default logical	yes

CEILING(A, KIND)

Returns the least integer greater than or equal to its argument.

A must be of type real.

KIND (optional)
must be a scalar integer initialization expression.

Class

Elemental function

Result Type and Attributes

- It is of type integer.
- If KIND is present, the kind type parameter is that specified by KIND; otherwise, the KIND type parameter is that of the default integer type.

Result Value

The result has a value equal to the least integer greater than or equal to A. The result is undefined if the result cannot be represented as an integer of the specified KIND.

Examples

```
CEILING(-3.7) has the value -3.
CEILING(3.7) has the value 4.
CEILING(1000.1, KIND=2) has the value 1 001, with a kind type parameter of two.
```

CHAR(I, KIND)

Returns the character in the given position of the collating sequence associated with the specified kind type parameter. It is the inverse of the function ICHAR.

I must be of type integer with a value in the range 0 ≤ I ≤ 127.

KIND (optional)
must be a scalar integer initialization expression.

Class

Elemental function

Result Type and Attributes

- Character of length one.
- If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of the default character type.

Result Value

- The result is the character in position I of the collating sequence associated with the specified kind type parameter.
- ICHAR (CHAR (I, KIND (C))) must have the value I for 0 ≤ I ≤ 127 and CHAR (ICHAR (C), KIND (C)) must have the value C for any representable character.

Examples

CHAR (88) has the value 'X'.

Notes:

1. XL Fortran supports only the ASCII collating sequence.

Specific Name	Argument Type	Result Type	Pass As Arg?
CHAR	any integer	default character	yes **1**

Notes:

1. The extension is the ability to pass the name as an argument.

CMPLX(X, Y, KIND)

Convert to complex type.

X must be of type integer, real, or complex.

Y (optional) must be of type integer or real. It must not be present if X is of type complex.

KIND (optional)
must be a scalar integer initialization expression.

Class

Elemental function

Result Type and Attributes

- It is of type complex.
- If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of the default real type.

Result Value

- If Y is absent and X is not complex, it is as if Y were present with the value zero.
- If Y is absent and X is complex, it is as if Y were present with the value AIMAG(X).
- CMPLX(X, Y, KIND) has the complex value whose real part is REAL(X, KIND) and whose imaginary part is REAL(Y, KIND).

Examples

CMPLX (-3) has the value (-3.0, 0.0).

Specific Name	Argument Type	Result Type	Pass As Arg?
CMPLX	default real	default complex	no

Related Information

"DCMPLX(X, Y)" on page 550, "QCMPLX(X, Y)" on page 607.

CONJG(Z)

Conjugate of a complex number.

Z must be of type complex.

Class

Elemental function

Result Type and Attributes

Same as Z.

Result Value

Given Z has the value (x, y), the result has the value (x, -y).

Examples

CONJG ((2.0, 3.0)) has the value (2.0, -3.0).

Specific Name	Argument Type	Result Type	Pass As Arg?
CONJG	default complex	default complex	yes
DCONJG	double complex	double complex	yes

Specific Name	Argument Type	Result Type	Pass As Arg?
QCONJG	COMPLEX(16)	COMPLEX(16)	yes

COS(X)

Cosine function.

X must be of type real or complex.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

- It has a value that approximates cos(X).
- If X is of type real, X is regarded as a value in radians.
- If X is of type complex, the real part of X is regarded as a value in radians.

Examples

COS (1.0) has the value 0.54030231 (approximately).

Specific Name	Argument Type	Result Type	Pass As Arg?
COS	default real	default real	yes
DCOS	double precision real	double precision real	yes
QCOS	REAL(16)	REAL(16)	yes
CCOS 1	default complex	default complex	yes
CDCOS 2	double complex	double complex	yes
ZCOS 2	double complex	double complex	yes
CQCOS 2	COMPLEX(16)	COMPLEX(16)	yes

COSD(X)

Cosine function. Argument in degrees.

X must be of type real.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

- It approximates cos(X), where X has a value in degrees.

Examples

COSD (45.0) has the value 0.7071067691.

Specific Name	Argument Type	Result Type	Pass As Arg?
COSD	default real	default real	yes
DCOSD	double precision real	double precision real	yes
QCOSD	REAL(16)	REAL(16)	yes

COSH(X)

Hyperbolic cosine function.

X must be of type real.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

The result value approximates cosh(X).

Examples

COSH (1.0) has the value 1.5430806 (approximately).

Specific Name	Argument Type	Result Type	Pass As Arg?
COSH 1	default real	default real	yes
DCOSH 2	double precision real	double precision real	yes
QCOSH 2	REAL(16)	REAL(16)	yes

COUNT(MASK, DIM)

Counts the number of true array elements in an entire logical array, or in each vector along a single dimension. Typically, the logical array is one that is used as a mask in another intrinsic.

MASK is a logical array.

DIM (optional)
: is an integer scalar in the range 1 ≤ DIM ≤ rank(MASK). The corresponding actual argument must not be an optional dummy argument.

Class

Transformational function

Result Value

If DIM is present, the result is an integer array of rank rank(MASK)-1. If DIM is missing, or if MASK has a rank of one, the result is a scalar of type integer.

Each element of the resulting array $(R(s_1, s_2, ..., s_{(DIM-1)}, s_{(DIM+1)}, ..., s_n))$ equals the number of elements that are true in MASK along the corresponding dimension $(s_1, s_2, ..., s_{(DIM-1)}, :, s_{(DIM+1)}, ..., s_n)$.

If MASK is a zero-sized array, the result equals zero.

Examples

```
! A is the array | T F F |, and B is the array | F F T |
!                | F T T |                     | T T T |

! How many corresponding elements in A and B
! are equivalent?
    RES = COUNT(A .EQV. B)        ! result RES is 3

! How many corresponding elements are equivalent
! in each column?
    RES = COUNT(A .EQV. B, DIM=1) ! result RES is (0,2,1)

! Same question, but for each row.
    RES = COUNT(A .EQV. B, DIM=2) ! result RES is (1,2)
```

CPU_TIME(TIME)

Returns the CPU time, in seconds, taken by the current process and, possibly, all the child processes in all of the threads. A call to **CPU_TIME** will give the processor time taken by the process from the start of the program. The time measured only accounts for the amount of time that the program is actually running, and not the time that a program is suspended or waiting.

TIME
: Is a scalar of type real. It is an **INTENT(OUT)** argument that is assigned an approximation to the processor time. The time is measured in seconds. The time returned by **CPU_TIME** is dependent upon the setting of the **XLFRTEOPTS** environment variable run-time option **cpu_time_type**. The valid settings for **cpu_time_type** are:

  usertime
  : The user time for the current process. For a

definition of user time, see the *AIX Performance and Tuning Guide*.

systime
The system time for the current process. For a definition of system time, see the *AIX Performance and Tuning Guide*.

alltime
The sum of the user and system time for the current process

total_usertime
The total user time for the current process. The total user time is the sum of the user time for the current process and the total user times for its child processes, if any.

total_systime
The total system time for the current process. The total system time is the sum of the system time for the current process and the total system times for its child processes, if any.

total_alltime
The total user and system time for the current process. The total user and system time is the sum of the user and system time for the current process and the total user and system times for their child processes, if any.

This is the default measure of time for **CPU_TIME** if you have not set the **cpu_time_type** run-time option.

You can set the **cpu_time_type** run-time option using the **setrteopts** procedure. Each change to the **cpu_time_type** setting will affect all subsequent calls to **CPU_TIME**.

Class

Subroutine

Examples

Example 1:

```
! The default value for cpu_time_type is used
REAL T1, T2
...         ! First chunk of code to be timed
CALL CPU_TIME(T1)
...         ! Second chunk of code to be timed
CALL CPU_TIME(T2)
print *, 'Time taken for first chunk of code: ', T1, 'seconds.'
print *, 'Time taken for both chunks of code: ', T2, 'seconds.'
print *, 'Time for second chunk of code was ', T2-T1, 'seconds.'
```

If you want to set the **cpu_time_type** run-time option to **usertime**, you would type the following command from a ksh or bsh command line:

```
export XLFRTEOPTS=cpu_time_type=usertime
```

Example 2:

```
! Use setrteopts to set the cpu_time_type run-time option as many times
! as you need to
CALL setrteopts ('cpu_time_type=alltime')
CALL stallingloop
CALL CPU_TIME(T1)
print *, 'The sum of the user and system time is', T1, 'seconds'.
CALL setrteopts ('cpu_time_type=usertime')
CALL stallingloop
CALL CPU_TIME(T2)
print *, 'The total user time from the start of the program is', T2, 'seconds'.
```

Related Information

- See the description of the "XLFRTEOPTS" environment variable in the *XL Fortran User's Guide* for more information.
- See the description of the "setrteopts" service and utility procedure on 664 and in the *XL Fortran User's Guide* for more information.

CSHIFT(ARRAY, SHIFT, DIM)

Shifts the elements of all vectors along a given dimension of an array. The shift is circular; that is, elements shifted off one end are inserted again at the other end.

ARRAY is an array of any type.

SHIFT must be a scalar integer if ARRAY has a rank of one; otherwise, it is a scalar integer or an integer expression of rank rank(ARRAY)-1.

DIM (optional)
is an integer scalar in the range 1 ≤ DIM ≤ rank(ARRAY). If absent, it defaults to 1.

Class

Transformational function

Result Value

The result is an array with the same shape and the same data type as ARRAY.

If SHIFT is a scalar, the same shift is applied to each vector. Otherwise, each vector ARRAY (s_1, s_2, ..., $s_{(DIM-1)}$, :, $s_{(DIM+1)}$, ..., s_n) is shifted according to the corresponding value in SHIFT (s_1, s_2, ..., $s_{(DIM-1)}$, $s_{(DIM+1)}$, ..., s_n)

The absolute value of SHIFT determines the amount of shift. The sign of SHIFT determines the direction of the shift:

Positive SHIFT
moves each element of the vector toward the beginning of the vector.

Negative SHIFT
moves each element of the vector toward the end of the vector.

Zero SHIFT does no shifting. The value of the vector remains unchanged.

Examples

```
! A is the array | A D G |
!                | B E H |
!                | C F I |

! Shift the first column down one, the second column
! up one, and leave the third column unchanged.
      RES = CSHIFT (A, SHIFT = (/-1,1,0/), DIM = 1)
! The result is | C E G |
!               | A F H |
!               | B D I |

! Do the same shifts as before, but on the rows
! instead of the columns.
      RES = CSHIFT (A, SHIFT = (/-1,1,0/), DIM = 2)
! The result is | G A D |
!               | E H B |
!               | C F I |
```

CVMGx(TSOURCE, FSOURCE, MASK)

The conditional vector merge functions (**CVMGM**, **CVMGN**, **CVMGP**, **CVMGT**, and **CVMGZ**) enable you to port existing code that contains these functions.

Calling them is very similar to calling

```
MERGE ( TSOURCE, FSOURCE, arith_expr .op. 0 )
or
MERGE ( TSOURCE, FSOURCE, logical_expr .op. .TRUE. )
```

Because the **MERGE** intrinsic is part of the Fortran 90 language, we recommend that you use it instead of these functions for any new programs.

TSOURCE is a scalar or array expression of type **LOGICAL**, **INTEGER**, or **REAL** and any kind except 1.

FSOURCE is a scalar or array expression with the same type and type parameters as TSOURCE.

MASK is a scalar or array expression of type **INTEGER** or **REAL** (for **CVMGM**, **CVMGN**, **CVMGP**, and **CVMGZ**) or **LOGICAL**

(for **CVMGT**), and any kind except 1. If it is an array, it must conform in shape to TSOURCE and FSOURCE.

If only one of TSOURCE and FSOURCE is typeless, the typeless argument acquires the type of the other argument. If both TSOURCE and FSOURCE are typeless, both arguments acquire the type of MASK. If MASK is also typeless, both TSOURCE and FSOURCE are treated as default integers. If MASK is typeless, it is treated as a default logical for the **CVMGT** function and as a default integer for the other **CVMGx** functions.

Class

Elemental function

Result Type and Attributes

Same as TSOURCE and FSOURCE.

Result Value

The function result is the value of either the first argument or second argument, depending on the result of the test performed on the third argument. If the arguments are arrays, the test is performed for each element of the MASK array, and the result may contain some elements from TSOURCE and some elements from FSOURCE.

Table 15. Result Values for CVMGx Intrinsic Procedures

Explanation	Function Return Value	Generic Name
Test for positive or zero	TSOURCE if MASK≥0 FSOURCE if MASK<0	CVMGP
Test for negative	TSOURCE if MASK<0 FSOURCE if MASK≥0	CVMGM
Test for zero	TSOURCE if MASK=0 FSOURCE if MASK≠0	CVMGZ
Test for nonzero	TSOURCE if MASK≠0 FSOURCE if MASK=0	CVMGN
Test for true	TSOURCE if MASK= .true. FSOURCE if MASK= .false.	CVMGT

DATE_AND_TIME(DATE, TIME, ZONE, VALUES)

Returns data from the real-time clock and the date in a form compatible with the representations defined in ISO 8601:1988.

DATE (optional)
must be scalar and of type default character, and must have a length of at least eight to contain the complete value. It is an INTENT(OUT) argument. Its leftmost eight characters are set to a value of the form CCYYMMDD, where CC is the century,

YY is the year within the century, MM is the month within the year, and DD is the day within the month. If no date is available, these characters are set to blank.

TIME (optional)
must be scalar and of type default character, and must have a length of at least ten in order to contain the complete value. It is an INTENT(OUT) argument. Its leftmost ten characters are set to a value of the form hhmmss.sss, where hh is the hour of the day, mm is the minutes of the hour, and ss.sss is the seconds and milliseconds of the minute. If no clock is available, they are set to blank.

ZONE (optional)
must be scalar and of type default character, and must have a length at least five in order to contain the complete value. It is an INTENT(OUT) argument. Its leftmost five characters are set to a value of the form ±hhmm, where hh and mm are the time difference with respect to Coordinated Universal Time (UTC) in hours and the parts of an hour[4] expressed in minutes, respectively. If no clock is available, they are set to blank.

VALUES (optional)
must be of type default integer and of rank one. It is an INTENT(OUT) argument. Its size must be at least eight. The values returned in VALUES are as follows:

VALUES(1)
is the year (for example, 1998), or -HUGE (0) if no date is available.

VALUES(2)
is the month of the year, or -HUGE (0) if no date is available.

VALUES(3)
is the day of the month, or -HUGE (0) if no date is available.

VALUES(4)
is the time difference with respect to Coordinated Universal Time (UTC) in minutes, or -HUGE (0) if this information is not available.

4. The value of **ZONE** may be incorrect if you have not set up the machine through the **smit chtz** fastpath, or if you are in a timezone not configurable through **smit**. You can manually set the **TZ** environment variable or use the **chtz** command to ensure the time zone is correctly set up. The format of the **TZ** variable is documented under the **/etc/environment** file in the *AIX Files Reference*.

VALUES(5)
is the hour of the day, in the range 0 to 23, or -HUGE (0) if there is no clock.

VALUES(6)
is the minutes of the hour, in the range 0 to 59, or -HUGE (0) if there is no clock.

VALUES(7)
is the seconds of the minute, in the range 0 to 60, or -HUGE (0) if there is no clock.

VALUES (8)
is the milliseconds of the second, in the range 0 to 999, or -HUGE (0) if there is no clock.

Class

Subroutine

Examples

The following program:

```
INTEGER DATE_TIME (8)
CHARACTER (LEN = 10) BIG_BEN (3)
CALL DATE_AND_TIME (BIG_BEN (1), BIG_BEN (2), &
                    BIG_BEN (3), DATE_TIME)
```

if executed in Geneva, Switzerland on 1985 April 12 at 15:27:35.5, would have assigned the value 19850412 to BIG_BEN(1), the value 152735.500 to BIG_BEN(2), the value +0100 to BIG_BEN(3), and the following values to DATE_TIME: 1985, 4, 12, 60, 15, 27, 35, 500.

Note that UTC is defined by CCIR Recommendation 460-2 (also known as Greenwich Mean Time).

DBLE(A)

Convert to double precision real type.

A must be of type integer, real, or complex.

Class

Elemental function

Result Type and Attributes

Double precision real.

Result Value

- If A is of type double precision real, DBLE(A) = A.

- If A is of type integer or real, the result has as much precision of the significant part of A as a double precision real datum can contain.
- If A is of type complex, the result has as much precision of the significant part of the real part of A as a double precision real datum can contain.

Examples

DBLE (-3) has the value -3.0D0.

Specific Name	Argument Type	Result Type	Pass As Arg?
DFLOAT	any integer	double precision real	no
DBLE	default real	double precision real	no
DBLEQ	REAL(16)	REAL(8)	no

DCMPLX(X, Y)

Convert to double complex type.

X must be of type integer, real, or complex.

Y (optional) must be of type integer or real. It must not be present if X is of type complex.

Class

Elemental function

Result Type and Attributes

It is of type double complex.

Result Value

- If Y is absent and X is not complex, it is as if Y were present with the value of zero.
- If Y is absent and X is complex, it is as if Y were present with the value AIMAG(X).
- DCMPLX(X, Y) has the complex value whose real part is REAL(X, KIND=8) and whose imaginary part is REAL(Y, KIND=8).

Examples

DCMPLX (-3) has the value (-3.0D0, 0.0D0).

Specific Name	Argument Type	Result Type	Pass As Arg?
DCMPLX	double precision real	double complex	no

Related Information

"CMPLX(X, Y, KIND)" on page 539, "QCMPLX(X, Y)" on page 607.

DIGITS(X)

Returns the number of significant digits for numbers whose type and kind type parameter are the same as the argument.

X must be of type integer or real. It may be scalar or array valued.

Class

Inquiry function

Result Type and Attributes

Default integer scalar.

Result Value

- If X is of type integer, the number of the significant digits of X is:

```
    type                    bits
-----------              ------
 integer(1)                07
 integer(2)                15
 integer(4)                31
 integer(8)                63
```

- If X is of type real, the number of significant bits of X is:

```
   type                    bits
----------              ------
real(4)                   24
real(8)                   53
real(16)                 106
```

Examples

DIGITS (X) = 63, where X is of type integer(8) (see "Data Representation Models" on page 521).

DIM(X, Y)

The difference X-Y if it is positive; otherwise zero.

X must be of type integer or real.

Y must be of the same type and kind type parameter as X.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

- If X > Y, the value of the result is X - Y.
- If X ≤ Y, the value of the result is zero.

Examples

DIM (-3.0, 2.0) has the value 0.0. DIM (-3.0, -4.0) has the value 1.0.

Specific Name	Argument Type	Result Type	Pass As Arg?
IDIM	any integer 1	same as argument	yes
DIM	default real	default real	yes
DDIM	double precision real	double precision real	yes
QDIM	REAL(16)	REAL(16)	yes

Notes:

1. The extension is the ability to specify a nondefault integer argument.

DOT_PRODUCT(VECTOR_A, VECTOR_B)

Computes the dot product on two vectors.

VECTOR_A is a vector with a numeric or logical data type.

VECTOR_B must be of numeric type if VECTOR_A is of numeric type and of logical type if VECTOR_A is of logical type. It must be the same size as VECTOR_A.

Class

Transformational function

Result Value

The result is a scalar whose data type depends on the data type of the two vectors, according to the rules in Table 3 on page 92 and Table 4 on page 98.

If either vector is a zero-sized array, the result equals zero when it has a numeric data type, and false when it is of type logical.

If VECTOR_A is of type integer or real, the result value equals SUM(VECTOR_A * VECTOR_B).

If VECTOR_A is of type complex, the result equals SUM(CONJG(VECTOR_A) * VECTOR_A).

If VECTOR_A is of type logical, the result equals ANY(VECTOR_A .AND. VECTOR_B).

Examples

```
! A is (/ 3, 1, -5 /), and B is (/ 6, 2, 7 /).
      RES = DOT_PRODUCT (A, B)
! calculated as
!    ( (3*6) + (1*2) + (-5*7) )
! = (   18  +    2  +  (-35) )
! =  -15
```

DPROD(X, Y)

Double precision real product.

X must be of type default real.

Y must be of type default real.

Class

Elemental function

Result Type and Attributes

Double precision real.

Result Value

The result has a value equal to the product of X and Y.

Examples

DPROD (-3.0, 2.0) has the value -6.0D0.

Specific Name	Argument Type	Result Type	Pass As Arg?
DPROD	default real	double precision real	yes
QPROD	double precision real	REAL(16)	yes

EOSHIFT(ARRAY, SHIFT, BOUNDARY, DIM)

Shifts the elements of all vectors along a given dimension of an array. The shift is end-off; that is, elements shifted off one end are lost, and copies of boundary elements are shifted in at the other end.

ARRAY is an array of any type.

SHIFT is a scalar of type integer if ARRAY has a rank of 1; otherwise, it is a scalar integer or an integer expression of rank rank(ARRAY)-1.

BOUNDARY (optional) is of the same type and type parameters as ARRAY. If ARRAY has a rank of 1,

	BOUNDARY must be scalar; otherwise, it is a scalar or an expression of rank rank(ARRAY)-1.
DIM (optional)	is an integer scalar in the range 1 ≤ DIM ≤ rank(ARRAY).

Class

Transformational function

Result Value

The result is an array with the same shape and data type as ARRAY.

The absolute value of SHIFT determines the amount of shift. The sign of SHIFT determines the direction of the shift:

Positive SHIFT	moves each element of the vector toward the beginning of the vector. If an element is taken off the beginning of a vector, its value is replaced by the corresponding value from BOUNDARY at the end of the vector.
Negative SHIFT	moves each element of the vector toward the end of the vector. If an element is taken off the end of a vector, its value is replaced by the corresponding value from boundary at the beginning of the vector.
Zero SHIFT	does no shifting. The value of the vector remains unchanged.

Result Value

If BOUNDARY is a scalar value, this value is used in all shifts.

If BOUNDARY is an array of values, the values of the array elements of BOUNDARY with subscripts (s_1, s_2, ..., $s_{(DIM-1)}$, $s_{(DIM+1)}$, ..., s_n) are used for that dimension.

If BOUNDARY is not specified, the following default values are used, depending on the data type of ARRAY:

character	'ƀ' (one blank)
logical	false
integer	0
real	0.0
complex	(0.0, 0.0)

Examples

```
! A is | 1.1 4.4 7.7 |, SHIFT is S=(/0, -1, 1/),
!      | 2.2 5.5 8.8 |
!      | 3.3 6.6 9.9 |
! and BOUNDARY is the array B=(/-0.1, -0.2, -0.3/).

! Leave the first column alone, shift the second
! column down one, and shift the third column up one.
RES = EOSHIFT (A, SHIFT = S, BOUNDARY = B, DIM = 1)
! The result is | 1.1 -0.2  8.8 |
!               | 2.2  4.4  9.9 |
!               | 3.3  5.5 -0.3 |

! Do the same shifts as before, but on the
! rows instead of the columns.
RES = EOSHIFT (A, SHIFT = S, BOUNDARY = B, DIM = 2)
! The result is |  1.1  4.4  7.7 |
!               | -0.2  2.2  5.5 |
!               |  6.6  9.9 -0.3 |
```

EPSILON(X)

Returns a positive model number that is almost negligible compared to unity in the model representing numbers of the same type and kind type parameter as the argument.

X must be of type real. It may be scalar or array valued.

Class

Inquiry function

Result Type and Attributes

Scalar of the same type and kind type parameter as X.

Result Value

The result is

2.0*ei*0$^{1 - \mathrm{DIGITS(X)}}$

where *ei* is the exponent indicator (E, D, or Q) depending on the type of X:

```
type                      EPSILON(X)
----                   ------------------
real(4)                   02E0 ** (-23)
real(8)                   02D0 ** (-52)
real(16)                  02Q0 ** (-105)
```

Examples

EPSILON (X) = 1.1920929E-07 for X of type real(4). See “Real Data Model” on page 522.

ERF(X)

Error function.

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

X must be of type real.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

- The result value approximates erf(X).
- The result is in the range $-1 \leq \mathrm{ERF}(X) \leq 1$

Examples

ERF (1.0) has the value 0.8427007794 (approximately).

Specific Name	Argument Type	Result Type	Pass As Arg?
ERF	default real	default real	yes
DERF	double precision real	double precision real	yes
QERF	REAL(16)	REAL(16)	yes

ERFC(X)

Complementary error function.

$$\mathrm{erfc}(x) = 1 - \mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2} dt$$

X must be of type real.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

- The result has a value equal to 1-ERF(X).
- The result is in the range 0 ≤ ERFC(X) ≤ 2

Examples

ERFC (1.0) has the value 0.1572992057 (approximately).

Specific Name	Argument Type	Result Type	Pass As Arg?
ERFC	default real	default real	yes
DERFC	double precision real	double precision real	yes
QERFC	REAL(16)	REAL(16)	yes

EXP(X)

Exponential.

X must be of type real or complex.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

- The result approximates e^x.
- If X is of type complex, its imaginary part is regarded as a value in radians.

Examples

EXP (1.0) has the value 2.7182818 (approximately).

Specific Name	Argument Type	Result Type	Pass As Arg?
EXP **1**	default real	default real	yes
DEXP **2**	double precision real	double precision real	yes
QEXP **2**	REAL(16)	REAL(16)	yes
CEXP **3a**	default complex	default complex	yes
CDEXP **3b**	double complex	double complex	yes
ZEXP **3b**	double complex	double complex	yes
CQEXP **3b**	COMPLEX(16)	COMPLEX(16)	yes

EXPONENT(X)

Returns the exponent part of the argument when represented as a model number.

X must be of type real.

Class

Elemental function

Result Type and Attributes

Default integer.

Result Value

- If X ≠ 0, the result is the exponent of X (which is always within the range of a default integer).
- If X = 0, the exponent of X is zero.

Examples

EXPONENT (10.2) = 4. See "Real Data Model" on page 522

FLOOR(A, KIND)

Returns the greatest integer less than or equal to its argument.

A must be of type real.

KIND (optional)
must be a scalar integer initialization expression.

Class

Elemental function

Result Type and Attributes

- It is of type integer.
- If KIND is present, the kind type parameter is that specified by KIND; otherwise, the KIND type parameter is that of the default integer type.

Result Value

The result has a value equal to the least integer greater than or equal to A. The result is undefined if the result cannot be represented as an integer of the specified KIND.

Examples

```
FLOOR(-3.7) has the value -4.
FLOOR(3.7) has the value 3.
FLOOR(1000.1, KIND=2) has the value 1 000, with a kind type parameter of two.
```

FMADD(A, X, Y)

Returns the result of a floating-point multiply–add.

A
: must be of type **REAL(8)**. If compiled on a PowerPC platform with **–qarch** set for PowerPC compilation, A may alternatively be of type **REAL(4)**.

X
: must be of the same type and kind type parameter as A.

Y
: must be of the same type and kind type parameter as A.

Class

Elemental function.

Result Type and Attributes

Same as A, X, and Y.

Result Value

The result has a value equal to A*X + Y.

Examples

The following example is only valid if compiled with the **–qarch** option set for PowerPC compilation, because A, B, C and RES1 are single precision reals.

```
REAL(4) :: A, B, C, RES1
REAL(8) :: D, E, F, RES2

RES1 = FMADD(A, B, C)
RES2 = FMADD(D, E, F)
END
```

Related Information

See the **–qarch** compiler option in the *User's Guide* for details about compiling for a PowerPC.

FMSUB(A, X, Y)

Returns the result of a floating-point multiply–subtract.

A
: must be of type **REAL(8)**. If compiled on a PowerPC platform with **–qarch** set for PowerPC compilation, A may alternatively be of type **REAL(4)**.

X
: must be of the same type and kind type parameter as A.

Y
: must be of the same type and kind type parameter as A.

Class

Elemental function.

Result Type and Attributes

Same as A, X, and Y.

Result Value

The result has a value equal to A*X – Y.

Related Information

See the **–qarch** compiler option in the *User's Guide* for details about compiling for a PowerPC.

FNMADD(A, X, Y)

Returns the result of a floating-point negative multiply–add.

A
: must be of type **REAL(8)**. If compiled on a PowerPC platform with **–qarch** set for PowerPC compilation, A may alternatively be of type **REAL(4)**.

X
: must be of the same type and kind type parameter as A.

Y
: must be of the same type and kind type parameter as A.

Class

Elemental function.

Result Type and Attributes

Same as A, X, and Y.

Result Value

The result has a value equal to –(A*X + Y).

Related Information

See the **–qarch** compiler option in the *User's Guide* for details about compiling for a PowerPC.

FNMSUB(A, X, Y)

Returns the result of a floating-point negative multiply–subtract.

A
: must be of type **REAL(8)**. If compiled on a PowerPC platform with **–qarch** set for PowerPC compilation, A may alternatively be of type **REAL(4)**.

X
: must be of the same type and kind type parameter as A.

Y
: must be of the same type and kind type parameter as A.

Class

Elemental function.

Result Type and Attributes

Same as A, X, and Y.

Result Value

The result has a value equal to –(A*X – Y).

Examples

In the following example, the result of FNMSUB is of type REAL(4). It is converted to REAL(8) and then assigned to RES.

```
REAL(4) :: A, B, C
REAL(8) :: RES

RES = FNMSUB(A, B, C)
END
```

Related Information

See the **–qarch** compiler option in the *User's Guide* for details about compiling for a PowerPC.

FRE(X)

Returns the result of a floating-point reciprocal operation.

This procedure can be run on a PowerPC platform only. You must specify a PowerPC architecture as part of the **–qarch** compiler option.

X must be of type **REAL(4)**.

Class

Elemental function.

Result Type and Attributes

Same as X.

Result Value

The result is a single precision estimate of 1/x.

Related Information

See the **–qarch** compiler option in the *User's Guide* for details about compiling for a PowerPC.

FRSQRTE(X)

Returns the result of a reciprocal square root operation.

This procedure can be run on a PowerPC platform only. You must specify a PowerPC architecture as part of the **–qarch** compiler option.

X must be of type **REAL(8)**.

Class

Elemental function.

Result Type and Attributes

Same as X.

Result Value

The result is a single precision estimate of the reciprocal of the square root of X.

Related Information

See the **–qarch** compiler option in the *User's Guide* for details about compiling for a PowerPC.

FSEL(X,Y,Z)

Returns the result of a floating-point selection operation. This result is determined by comparing the value of X with zero.

This procedure can be run on a PowerPC platform only. You must specify a PowerPC architecture as part of the **–qarch** compiler option.

X must be of type **REAL(4)** or **REAL(8)**.

Class

Elemental function.

Result Type and Attributes

Same as X, Y and Z.

Result Value

- If the value of X is greater than or equal to zero, then the value of Y is returned.
- If the value of X is smaller than zero or is a NaN, then the value of Z is returned.

A zero value is considered unsigned. That is, both +0 and -0 are equal to zero.

Related Information

See the **–qarch** compiler option in the *User's Guide* for details about compiling for a PowerPC.

FRACTION(X)

Returns the fractional part of the model representation of the argument value.

X must be of type real.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

The result is:

$X * (2.0^{-EXPONENT(X)})$

Examples

$FRACTION(10.2) = 2^{-4} * 10.2 \approx 0.6375$

GAMMA(X)

Gamma function.

$$\Gamma(x) = \int_0^\infty u^{x-1} e^{-u} du$$

X must be of type real.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

The result has a value that approximates Γ(X).

Examples

GAMMA (1.0) has the value 1.0.

GAMMA (10.0) has the value 362880.0 (approximately).

Specific Name	Argument Type	Result Type	Pass As Arg?
GAMMA 1	default real	default real	yes
DGAMMA 2	double precision real	double precision real	yes
QGAMMA 3	REAL(16)	REAL(16)	yes

GETENV(NAME, VALUE)

Determines the value of the specified environment variable.

NAME
: is a character string that identifies the name of the operating-system environment variable. The string is case-significant. It is an INTENT(IN) argument that must be scalar of type default character.

VALUE
: holds the value of the environment variable when the subroutine returns. It is an INTENT(OUT) argument that must be scalar of type default character.

Class

Subroutine

Result Value

The result is returned in the VALUE argument, not as a function result variable.

If the environment variable specified in the NAME argument does not exist, the VALUE argument contains blanks.

Examples

```
      CHARACTER (LEN=16)   ENVDATA
      CALL GETENV('HOME', VALUE=ENVDATA)
! Print the value.
      PRINT *, ENVDATA
! Show how it is blank-padded on the right.
      WRITE(*, '(Z32)') ENVDATA
      END
```

The following is sample output generated by the above program:

```
 /home/mark
2F686F6D652F6D61726B202020202020
```

Related Information

See the **getenv** subroutine in the *Technical Reference: Base Operating System and Extensions Volume 1* for details about the operating-system-level implementation.

HFIX(A)

Convert from **REAL(4)** to **INTEGER(2)**.

This procedure is a specific function, not a generic function.

A must be of type **REAL(4)**.

Class

Elemental function

Result Type and Attributes

An **INTEGER(2)** scalar or array.

Result Value

- If |A| < 1, INT (A) has the value 0.
- If |A| ≥ 1, INT (A) is the integer whose magnitude is the largest integer that does not exceed the magnitude of A and whose sign is the same as the sign of A.
- The result is undefined if the result cannot be represented in an **INTEGER(2)**.

Examples

HFIX (-3.7) has the value -3.

Specific Name	Argument Type	Result Type	Pass As Arg?
HFIX	REAL(4)	INTEGER(2)	no

HUGE(X)

Returns the largest number in the model representing numbers of the same type and kind type parameter as the argument.

X must be of type integer or real. It may be scalar or array valued.

Class

Inquiry function

Result Type and Attributes

Scalar of the same type and kind type parameter as X.

Result Value

- If X is of any integer type, the result is:
  $2^{DIGITS(X)} - 1$
- If X is of any real type, the result is:
  $(1.0 - 2.0^{-DIGITS(X)}) *$
  $(2.0^{MAXEXPONENT(X)})$

Examples

HUGE (X) = (1D0 - 2D0**-53) * (2D0**1024) for X of type real(8).

HUGE (X) = (2**63) - 1 for X of type integer(8).

See "Data Representation Models" on page 521.

IACHAR(C)

Returns the position of a character in the ASCII collating sequence.

C must be of type default character and of length one.

Class

Elemental function

Result Type and Attributes

Default integer.

Result Value

- If C is in the collating sequence defined by the codes specified in ISO 646:1983 (International Reference Version), the result is the position of C in that sequence and satisfies the inequality (0 ≤ IACHAR (C) ≤ 127). An undefined value is returned if C is not in the ASCII collating sequence.
- The results are consistent with the LGE, LGT, LLE, and LLT lexical comparison functions. For example, LLE (C, D) is true, so IACHAR (C) .LE. IACHAR (D) is true too.

Examples

IACHAR ('X') has the value 88.

IAND(I, J)

Performs a logical AND.

I must be of type integer.

J must be of type integer with the same kind type parameter as I.

Class

Elemental function

Result Type and Attributes

Same as I.

Result Value

The result has the value obtained by combining I and J bit-by-bit according to the following table:

```
I   J   IAND (I,J)
------------------
1   1       1
1   0       0
0   1       0
0   0       0
```

The bits are numbered from 0 to BIT_SIZE(I)-1, from right to left.

Examples

IAND (1, 3) has the value 1. See "Integer Bit Model" on page 521.

Specific Name	Argument Type	Result Type	Pass As Arg?
IAND	any integer	same as argument	yes
AND	any integer	same as argument	yes

IBCLR(I, POS)

Clears one bit to zero.

I must be of type integer.

POS must be of type integer. It must be nonnegative and less than BIT_SIZE (I).

Class

Elemental function

Result Type and Attributes

Same as I.

Result Value

The result has the value of the sequence of bits of I, except that bit POS of I is set to zero.

The bits are numbered from 0 to BIT_SIZE(I)-1, from right to left.

Examples

IBCLR (14, 1) has the result 12.

If V has the value (/1, 2, 3, 4/), the value of IBCLR (POS = V, I = 31) is (/29, 27, 23, 15/).

See "Integer Bit Model" on page 521.

Specific Name	Argument Type	Result Type	Pass As Arg?
IBCLR	any integer	same as argument	yes

IBITS(I, POS, LEN)

Extracts a sequence of bits.

I must be of type integer.

POS must be of type integer. It must be nonnegative and POS + LEN must be less than or equal to BIT_SIZE (I).

LEN must be of type integer and nonnegative.

Class

Elemental function

Result Type and Attributes

Same as I.

Result Value

The result has the value of the sequence of LEN bits in I beginning at bit POS, right-adjusted and with all other bits zero.

The bits are numbered from 0 to BIT_SIZE(I)-1, from right to left.

Examples

IBITS (14, 1, 3) has the value 7. See “Integer Bit Model” on page 521.

Specific Name	Argument Type	Result Type	Pass As Arg?
IBITS	any integer	same as argument	yes

IBSET(I, POS)

Sets one bit to one.

I must be of type integer.

POS must be of type integer. It must be nonnegative and less than BIT_SIZE (I).

Class

Elemental function

Result Type and Attributes

Same as I.

Result Value

The result has the value of the sequence of bits of I, except that bit POS of I is set to one.

The bits are numbered from 0 to BIT_SIZE(I)-1, from right to left.

Examples

IBSET (12, 1) has the value 14.

If V has the value (/1, 2, 3, 4/), the value of IBSET (POS = V, I = 0) is (/2, 4, 8, 16/).

See “Integer Bit Model” on page 521.

Specific Name	Argument Type	Result Type	Pass As Arg?
IBSET	any integer	same as I	yes

ICHAR(C)

Returns the position of a character in the collating sequence associated with the kind type parameter of the character.

C must be of type character and of length one. Its value must be that of a representable character.

Class

Elemental function

Result Type and Attributes

Default integer.

Result Value

- The result is the position of C in the collating sequence associated with the kind type parameter of C and is in the range 0 ≤ ICHAR (C) ≤ 127.
- For any representable characters C and D, C .LE. D is true if and only if ICHAR (C) .LE. ICHAR (D) is true and C .EQ. D is true if and only if ICHAR (C) ·EQ· ICHAR (D) is true.

Examples

ICHAR ('X') has the value 88 in the ASCII collating sequence.

Notes:

1. XL Fortran supports only the ASCII collating sequence.

Specific Name	Argument Type	Result Type	Pass As Arg?
ICHAR	default character	default integer	yes **1**

Notes:

1. The extension is the ability to pass the name as an argument.

IEOR(I, J)

Performs an exclusive OR.

I must be of type integer.

J must be of type integer with the same kind type parameter as I.

Class

Elemental function

Result Type and Attributes

Same as I.

Result Value

The result has the value obtained by combining I and J bit-by-bit according to the following truth table:

```
I   J  IEOR (I,J)
-----------------
1   1      0
1   0      1
0   1      1
0   0      0
```

The bits are numbered 0 to BIT_SIZE(I)-1, from right to left.

Examples

IEOR (1, 3) has the value 2. See "Integer Bit Model" on page 521.

Specific Name	Argument Type	Result Type	Pass As Arg?
IEOR	any integer	same as argument	yes
XOR	any integer	same as argument	yes

ILEN(I)

Returns one less than the length, in bits, of the twos complement representation of an integer.

I is of type integer

Class

Elemental function

Result Type and Attributes

Same as I.

Result Value

- If I is negative, ILEN(I)=CEILING(LOG2(-I))
- If I is nonnegative, ILEN(I)=CEILING(LOG2(I+1))

Examples

```
I=ILEN(4)  ! 3
J=ILEN(-4) ! 2
```

IMAG(Z)

Identical to **AIMAG**.

Related Information

"AIMAG(Z), IMAG(Z)" on page 527.

INDEX(STRING, SUBSTRING, BACK)

Returns the starting position of a substring within a string.

STRING must be of type character.

SUBSTRING must be of type character with the same kind type parameter as STRING.

BACK (optional)
must be of type logical.

Class

Elemental function

Result Type and Attributes

Default integer.

Result Value

- Case (i): If BACK is absent or present with the value .FALSE., the result is the minimum positive value of I such that STRING (I : I + LEN (SUBSTRING) - 1) = SUBSTRING or zero if there is no such value. Zero is returned if LEN (STRING) < LEN (SUBSTRING). One is returned if LEN (SUBSTRING) = 0.
- Case (ii): If BACK is present with the value .TRUE., the result is the maximum value of I less than or equal to LEN (STRING) - LEN (SUBSTRING) + 1, such that STRING (I : I + LEN (SUBSTRING) - 1) = SUBSTRING or zero if there is no such value. Zero is returned if LEN (STRING) < LEN (SUBSTRING) and LEN (STRING) + 1 is returned if LEN (SUBSTRING) = 0.

Examples

INDEX ('FORTRAN', 'R') has the value 3.

INDEX ('FORTRAN', 'R', BACK = .TRUE.) has the value 5.

Specific Name	Argument Type	Result Type	Pass As Arg?
INDEX	default character	default integer	yes **1**

Note: When this specific name is passed as an argument, the procedure can only be referenced without the **BACK** optional argument.

INT(A, KIND)

Convert to integer type.

A must be of type integer, real, or complex.

KIND (optional)
must be a scalar integer initialization expression.

Class

Elemental function

Result Type and Attributes

- Integer.
- If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of the default integer type.

Result Value

- Case (i): If A is of type integer, INT (A) = A.
- Case (ii): If A is of type real, there are two cases: if $|A| < 1$, INT (A) has the value 0; if $|A| \geq 1$, INT (A) is the integer whose magnitude is the largest integer that does not exceed the magnitude of A and whose sign is the same as the sign of A.
- Case (iii): If A is of type complex, INT (A) is the value obtained by applying the case (ii) rule to the real part of A.
- The result is undefined if it cannot be represented in the specified integer type.

Examples

INT (-3.7) has the value -3.

Specific Name	Argument Type	Result Type	Pass As Arg?
INT	default real	default integer	no
IDINT	double precision real	default integer	no
IFIX	default real	default integer	no
IQINT	REAL(16)	default integer	no

IOR(I, J)

Performs an inclusive OR.

I must be of type integer.

J must be of type integer with the same kind type parameter as I.

Class

Elemental function

Result Type and Attributes

Same as I.

Result Value

The result has the value obtained by combining I and J bit-by-bit according to the following truth table:

```
I   J   IOR (I,J)
-----------------
1   1      1
1   0      1
0   1      1
0   0      0
```

The bits are numbered 0 to BIT_SIZE(I)-1, from right to left.

Examples

IOR (1, 3) has the value 3. See “Integer Bit Model” on page 521.

Specific Name	Argument Type	Result Type	Pass As Arg?
IOR	any integer	same as argument	yes
OR	any integer	same as argument	yes

ISHFT(I, SHIFT)

Performs a logical shift.

I must be of type integer.

SHIFT must be of type integer. The absolute value of SHIFT must be less than or equal to BIT_SIZE (I).

Class

Elemental function

Result Type and Attributes

Same as I.

Result Value

- The result has the value obtained by shifting the bits of I by SHIFT positions.
- If SHIFT is positive, the shift is to the left; if SHIFT is negative, the shift is to the right; and, if SHIFT is zero, no shift is performed.
- Bits shifted out from the left or from the right, as appropriate, are lost.
- Vacated bits are filled with zeros.
- The bits are numbered 0 to BIT_SIZE(I)-1, from right to left.

Examples

ISHFT (3, 1) has the result 6. See “Integer Bit Model” on page 521.

Specific Name	Argument Type	Result Type	Pass As Arg?
ISHFT	any integer	same as argument	yes

ISHFTC(I, SHIFT, SIZE)

Performs a circular shift of the rightmost bits; that is, bits shifted off one end are inserted again at the other end.

I
: must be of type integer.

SHIFT
: must be of type integer. The absolute value of SHIFT must be less than or equal to SIZE.

SIZE (optional)
: must be of type integer. The value of SIZE must be positive and must not exceed BIT_SIZE (I). If SIZE is absent, it is as if it were present with the value of BIT_SIZE (I).

Class

Elemental function

Result Type and Attributes

Same as I.

Result Value

The result has the value obtained by shifting the SIZE rightmost bits of I circularly by SHIFT positions. If SHIFT is positive, the shift is to the left; if SHIFT is negative, the shift is to the right; and, if SHIFT is zero, no shift is performed. No bits are lost. The unshifted bits are unaltered.

The bits are numbered 0 to BIT_SIZE(I)-1, from right to left.

Examples

ISHFTC (3, 2, 3) has the value 5. See "Integer Bit Model" on page 521.

Specific Name	Argument Type	Result Type	Pass As Arg?
ISHFTC	any integer	same as argument	yes **1**

Notes:

1. When this specific name is passed as an argument, the procedure can only be referenced with all three arguments.

KIND(X)

Returns the value of the kind type parameter of X.

X
: may be of any intrinsic type.

Class

Inquiry function

Result Type and Attributes

Default integer scalar.

Result Value

The result has a value equal to the kind type parameter value of X.

Kind type parameters supported by XL Fortran are defined in "Intrinsic Types" on page 29.

Examples

KIND (0.0) has the kind type parameter value of the default real type.

LBOUND(ARRAY, DIM)

Returns the lower bound of each dimension in an array, or the lower bound of a specified dimension.

ARRAY
: is the array whose lower bounds you want to determine. Its bounds must be defined; that is, it must not be a disassociated pointer or an allocatable array that is not allocated.

DIM (optional)
: is an integer scalar in the range 1 ≤ DIM ≤ rank(ARRAY). The corresponding actual argument must not be an optional dummy argument.

Class

Inquiry function

Result Type and Attributes

Default integer.

If DIM is present, the result is a scalar. If DIM is not present, the result is a one-dimensional array with one element for each dimension in ARRAY.

Result Value

Each element in the result corresponds to a dimension of array.

- If ARRAY is a whole array or array structure component, LBOUND(ARRAY, DIM) is equal to the lower bound for subscript DIM of ARRAY.

  The only exception is for a dimension that is zero-sized and ARRAY is not an assumed-size array of rank DIM, In such a case, the corresponding element in the result is one regardless of the value declared for the lower bound.
- If ARRAY is an array section or expression that is not a whole array or array structure component, each element has the value one.

Examples

```
        REAL A(1:10, -4:5, 4:-5)

        RES=LBOUND( A )
! The result is (/ 1, -4, 1 /).
```

```
       RES=LBOUND( A(:,:,:) )
       RES=LBOUND( A(4:10,-4:1,:) )
! The result in both cases is (/ 1, 1, 1 /)
! because the arguments are array sections.
```

LEADZ(I)

Returns the number of leading zero-bits in the binary representation of an integer.

I must be of type integer.

Class

Elemental function

Result Type and Attributes

Same as I.

Result Value

The result is the count of zero-bits to the left of the leftmost one-bit for an integer.

Examples

```
I = LEADZ(0_4)  ! I=32
J = LEADZ(4_4)  ! J=29
K = LEADZ(-1_4) ! K=0
```

LEN(STRING)

Returns the length of a character entity. The argument to this function need not be defined.

STRING must be of type character. It may be scalar or array valued.

Class

Inquiry function

Result Type and Attributes

Default integer scalar.

Result Value

The result has a value equal to the number of characters in STRING if it is scalar or in an element of STRING if it is array valued.

Examples

If C is declared by the statement

```
CHARACTER (11) C(100)
```

LEN (C) has the value 11.

Specific Name	Argument Type	Result Type	Pass As Arg?
LEN	default character	default integer	yes **1**

Notes:

1. The extension is the ability to pass the name as an argument.

LEN_TRIM(STRING)

Returns the length of the character argument without counting trailing blank characters.

STRING must be of type character.

Class

Elemental function

Result Type and Attributes

Default integer.

Result Value

The result has a value equal to the number of characters remaining after any trailing blanks in STRING are removed. If the argument contains no nonblank characters, the result is zero.

Examples

LEN_TRIM ('ƀAƀBƀ') has the value 4. LEN_TRIM ('ƀƀ') has the value 0.

LGAMMA(X)

Log of gamma function.

$$\log_e \Gamma(x) = \log_e \int_0^{\infty} u^{x-1} e^{-u} du$$

X must be of type real.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

The result has a value equal to $\log_e \Gamma(X)$.

Examples

LGAMMA (1.0) has the value 0.0.

LGAMMA (10.0) has the value 12.80182743 (approximately).

Specific Name	Argument Type	Result Type	Pass As Arg?
LGAMMA	default real	default real	yes
LGAMMA	double precision real	double precision real	yes
ALGAMA 1	default real	default real	yes
DLGAMA 2	double precision real	double precision real	yes
QLGAMA 3	REAL(16)	REAL(16)	yes

X must satisfy the inequality:

1. $0 < X \leq 4.0850E36$.
2. $2.3561D\text{-}304 \leq X \leq 2^{1014}$.
3. $2.3561Q\text{-}304 \leq X \leq 2^{1014}$.

LGE(STRING_A, STRING_B)

Test whether a string is lexically greater than or equal to another string, based on the ASCII collating sequence.

STRING_A must be of type default character.

STRING_B must be of type default character.

Class

Elemental function

Result Type and Attributes

Default logical.

Result Value

- If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string.
- If either string contains a character not in the ASCII character set, the result is undefined.
- The result is true if the strings are equal or if STRING_A follows STRING_B in the ASCII collating sequence; otherwise, the result is false. Note that the result is true if both STRING_A and STRING_B are of zero length.

Examples

LGE ('ONE', 'TWO') has the value .FALSE..

Specific Name	Argument Type	Result Type	Pass As Arg?
LGE	default character	default logical	yes **1**

Notes:

1. The extension is the ability to pass the name as an argument.

LGT(STRING_A, STRING_B)

Test whether a string is lexically greater than another string, based on the ASCII collating sequence.

STRING_A must be of type default character.

STRING_B must be of type default character.

Class

Elemental function

Result Type and Attributes

Default logical.

Result Value

- If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string.
- If either string contains a character not in the ASCII character set, the result is undefined.
- The result is true if STRING_A follows STRING_B in the ASCII collating sequence; otherwise, the result is false. Note that the result is false if both STRING_A and STRING_B are of zero length.

Examples

LGT ('ONE', 'TWO') has the value .FALSE..

Specific Name	Argument Type	Result Type	Pass As Arg?
LGT	default character	default logical	yes **1**

Notes:

1. The extension is the ability to pass the name as an argument.

LLE(STRING_A, STRING_B)

Test whether a string is lexically less than or equal to another string, based on the ASCII collating sequence.

STRING_A must be of type default character.

STRING_B must be of type default character.

Class

Elemental function

Result Type and Attributes

Default logical.

Result Value

- If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string.
- If either string contains a character not in the ASCII character set, the result is undefined.
- The result is true if the strings are equal or if STRING_A precedes STRING_B in the ASCII collating sequence; otherwise, the result is false. Note that the result is true if both STRING_A and STRING_B are of zero length.

Examples

LLE ('ONE', 'TWO') has the value .TRUE..

Specific Name	Argument Type	Result Type	Pass As Arg?
LLE	default character	default logical	yes 1

Notes:

1. The extension is the ability to pass the name as an argument.

LLT(STRING_A, STRING_B)

Test whether a string is lexically less than another string, based on the ASCII collating sequence.

STRING_A must be of type default character.

STRING_B must be of type default character.

Class

Elemental function

Result Type and Attributes

Default logical.

Result Value

- If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string.
- If either string contains a character not in the ASCII character set, the result is undefined.

- The result is true if STRING_A precedes STRING_B in the ASCII collating sequence; otherwise, the result is false. Note that the result is false if both STRING_A and STRING_B are of zero length.

Examples

LLT ('ONE', 'TWO') has the value .TRUE..

Specific Name	Argument Type	Result Type	Pass As Arg?
LLT	default character	default logical	yes 1

Notes:

1. The extension is the ability to pass the name as an argument.

LOC(X)

Returns the address of X that can then be used to define an integer **POINTER**.

X is the data object whose address you want to find. It must not be an undefined or disassociated pointer or a parameter. If it is a zero-sized array, it must be storage associated with a non-zero-sized storage sequence. If it is an array section, the storage of the array section must be contiguous.

Class

Inquiry function

Result Type and Attributes

The result is of type INTEGER(4) in 32-bit mode and of type INTEGER(8) in 64-bit mode.

Result Value

The result is the address of the data object, or, if X is a pointer, the address of the associated target. The result is undefined if the argument is not valid.

Examples

```
INTEGER A,B
POINTER (P,I)

P=LOC(A)
P=LOC(B)
END
```

LOG(X)

Natural logarithm.

X must be of type real or complex.

- If X is real, its value must be greater than zero.

- If X is complex, its value must not be zero.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

- It has a value approximating $\log_e X$.
- For complex arguments, LOG ((a,b)) approximates LOG (ABS((a,b))) + ATAN2((b,a)).

  If the argument type is complex, the result is the principal value of the imaginary part ω in the range $-\pi < \omega \le \pi$. If the real part of the argument is less than zero and its imaginary part is zero, the imaginary part of the result approximates π.

Examples

LOG (10.0) has the value 2.3025851 (approximately).

Specific Name	Argument Type	Result Type	Pass As Arg?
ALOG	default real	default real	yes
DLOG	double precision real	double precision real	yes
QLOG	REAL(16)	REAL(16)	yes
CLOG	default complex	default complex	yes
CDLOG	double complex	double complex	yes
ZLOG	double complex	double complex	yes
CQLOG	COMPLEX(16)	COMPLEX(16)	yes

LOG10(X)

Common logarithm.

X must be of type real. The value of X must be greater than zero.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

The result has a value equal to $\log_{10} X$.

Examples

LOG10 (10.0) has the value 1.0.

Specific Name	Argument Type	Result Type	Pass As Arg?
ALOG10	default real	default real	yes
DLOG10	double precision real	double precision real	yes
QLOG10	REAL(16)	REAL(16)	yes

LOGICAL(L, KIND)

Converts between objects of type logical with different kind type parameter values.

L must be of type logical.

KIND (optional)
must be a scalar integer initialization expression.

Class

Elemental function

Result Type and Attributes

- Logical.
- If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of the default logical type.

Result Value

The value is that of L.

Examples

LOGICAL (L .OR. .NOT. L) has the value .TRUE. and is of type default logical, regardless of the kind type parameter of the logical variable L.

LSHIFT(I, SHIFT)

Performs a logical shift to the left.

I must be of type integer.

SHIFT must be of type integer. It must be non-negative and less than or equal to BIT_SIZE(I).

Class

Elemental function

Result Type and Attributes

Same as I.

Result Value

- The result has the value obtained by shifting the bits of I by SHIFT positions to the left.
- Vacated bits are filled with zeros.
- The bits are numbered 0 to BIT_SIZE(I)-1, from right to left.

Examples

LSHIFT (3, 1) has the result 6.

LSHIFT (3, 2) has the result 12.

Specific Name	Argument Type	Result Type	Pass As Arg?
LSHIFT	any integer	same as argument	yes

MATMUL(MATRIX_A, MATRIX_B, MINDIM)

Performs a matrix multiplication.

MATRIX_A
is an array with a rank of one or two and a numeric or logical data type.

MATRIX_B
is an array with a rank of one or two and a numeric or logical data type. It can be a different numeric type than MATRIX_A, but you cannot use one numeric matrix and one logical matrix.

MINDIM (optional)
is an integer that determines whether to do the matrix multiplication using the Winograd variation of the Strassen algorithm, which may be faster for large matrices. The algorithm recursively splits the operand matrices into four roughly equal parts, until any submatrix extent is less than MINDIM.

Note: Strassen's method is not stable for certain row or column scalings of the input matrices. Therefore, for MATRIX_A and MATRIX_B with divergent exponent values, Strassen's method may give inaccurate results.

The significance of the value of MINDIM is:

<=0
does not use the Strassen algorithm at all. This is the default.

1
is reserved for future use.

>1
recursively applies the Strassen algorithm as long as the smallest extent of all dimensions in the argument

arrays is greater than or equal to this value. To achieve optimal performance you should experiment with the value of **MINDIM** as the optimal value depends on your machine configuration, available memory, and the size, type, and kind type of the arrays.

By default, **MATMUL** employs the conventional O(N**3) method of matrix multiplication.

If you link the **libxlf90_r.a** library, a parallel implementation of matrix multiplication is employed, which improves performance on SMP machines.

If you link the **libxlf90.a** or **libxlf90_t.a** library, the Winograd variation of the O(N**2.81) Strassen method is employed under these conditions:

1. MATRIX_A and MATRIX_B are both integer, both real, or both complex and have the same kind type.
2. The program can allocate the needed temporary storage, enough to hold approximately (2/3)*(N**2) elements for square matrices of extent N.
3. The MINDIM argument is less than or equal to the smallest of all extents of MATRIX_A and MATRIX_B.

At least one of the arguments must be of rank two. The size of the first or only dimension of MATRIX_B must be equal to the last or only dimension of MATRIX_A.

Class

Transformational function

Result Value

The result is an array. If one of the arguments is of rank one, the result has a rank of one. If both arguments are of rank two, the result has a rank of two.

The data type of the result depends on the data type of the arguments, according to the rules in Table 3 on page 92 and Table 4 on page 98.

If MATRIX_A and MATRIX_B have a numeric data type, the array elements of the result are:

Value of Element (i,j) = SUM((row i of MATRIX_A) * (column j of MATRIX_B))

If MATRIX_A and MATRIX_B are of type logical, the array elements of the result are:

Value of Element (i,j) = ANY((row i of MATRIX_A) .AND. (column j of MATRIX_B))

Examples

```
! A is the array  | 1 2 3 |, B is the array | 7 10 |
!                 | 4 5 6 |                 | 8 11 |
!                                           | 9 12 |
   RES = MATMUL(A, B)
! The result is |  50   68 |
!               | 122  167 |


! HUGE_ARRAY and GIGANTIC_ARRAY in this example are
! large arrays of real or complex type, so the operation
! might be faster with the Strassen algorithm.

   RES = MATMUL(HUGE_ARRAY, GIGANTIC_ARRAY, MINDIM=196)
```

Related Information

The numerical stability of Strassen's method for matrix multiplication is discussed in:

"Exploiting Fast Matrix Multiplication Within the Level 3 BLAS", Nicholas J. Higham, *ACM Transactions on Mathematical Software*, Vol. 16, No. 4, December 1990.

"GEMMW: A portable level 3 BLAS Winograd variant of Strassen's matrix-matrix multiply algorithm", Douglas, C. C., Heroux, M., Slishman, G., and Smith, R. M., *Journal of Computational Physics*, Vol. 110, No. 1, January 1994, pages 1-10.

MAX(A1, A2, A3, ...)

Maximum value.

- **A3, ...** are optional arguments. Any array that is itself an optional dummy argument must not be passed as an optional argument to this function unless it is present in the calling procedure.
- All the arguments must have the same type, either integer or real, and they all must have the same kind type parameter.

Class

Elemental function

Result Type and Attributes

Same as the arguments. (Some specific functions return results of a particular type.)

Result Value

The value of the result is that of the largest argument.

Examples

MAX (-9.0, 7.0, 2.0) has the value 7.0.

If you evaluate MAX (10, 3, A), where A is an optional array argument in the calling procedure, PRESENT(A) must be true in the calling procedure.

Specific Name	Argument Type	Result Type	Pass As Arg?
AMAX0	any integer **1**	default real	no
AMAX1	default real	default real	no
DMAX1	double precision real	double precision real	no
QMAX1	REAL(16)	REAL(16)	no
MAX0	any integer **1**	same as argument	no
MAX1	any real **2**	default integer	no

Notes:

1. The extension is the ability to specify a nondefault integer argument.
2. The extension is the ability to specify a nondefault real argument.

MAXEXPONENT(X)

Returns the maximum exponent in the model representing numbers of the same type and kind type parameter as the argument.

X must be of type real. It may be scalar or array valued.

Class

Inquiry function

Result Type and Attributes

Default integer scalar.

Result Value

The result is the following:

```
   type            MAXEXPONENT
----------         -----------
  real(4)              128
  real(8)             1024
  real(16)            1024
```

Examples

```
MAXEXPONENT(X) = 128 for X of type real(4).
```

See “Real Data Model” on page 522.

MAXLOC(ARRAY, DIM, MASK) or MAXLOC(ARRAY, MASK)

Locates the first element of an array along a dimension that has the maximum value of all elements corresponding to the true values of the mask. MAXLOC will return the index referable to the position of the element using a positive integer.

ARRAY
is an array of type integer or real.

DIM
is a scalar integer in the range 1≤DIM≤rank(ARRAY).

MASK (optional)
is of type logical and conforms to ARRAY in shape. If it is absent, the default mask evaluation is `.TRUE.`; that is, the entire array is evaluated.

Class

Transformational function

Result Type and Attributes

If DIM is absent, the result is an integer array of rank one with a size equal to the rank of ARRAY. If DIM is present, the result is an integer array of rank rank(ARRAY)-1, and the shape is (s_1, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n), where n is the rank of ARRAY.

If there is no maximum value, perhaps because the array is zero-sized or the mask array has all `.FALSE.` values or there is no DIM argument, the return value is a zero-sized one-dimensional entity. If DIM is present, the result shape depends on the rank of ARRAY.

Result Value

The result indicates the subscript of the location of the maximum masked element of ARRAY. If more than one element is equal to this maximum value, the function finds the location of the first (in array element order). If DIM is specified, the result indicates the location of the maximum masked element along each vector of the dimension.

Because both DIM and MASK are optional, various combinations of arguments are possible. When the **-qintlog** option is specified with two arguments, the second argument refers to one of the following:

- MASK if it is an array of type integer, logical, byte or typeless
- DIM if it is a scalar of type integer, byte or typeless
- MASK if it is a scalar of type logical

The addition of the DIM argument modifies the behavior from XL Fortran Version 3.

Examples

```
! A is the array  |  4  9  8 -8 |
!                 |  2  1 -1  5 |
!                 |  9  4 -1  9 |
!                 | -7  5  7 -3 |

! Where is the largest element of A?
       RES = MAXLOC(A)
! The result is | 3 1 | because 9 is located at A(3,1).
! Although there are other 9s, A(3,1) is the first in
! column-major order.

! Where is the largest element in each column of A
! that is less than 7?
       RES = MAXLOC(A, DIM = 1, MASK = A .LT. 7)
! The result is | 1 4 2 2 | because these are the corresponding
! row locations of the largest value in each column
! that are less than 7 (the values being 4,5,-1,5).
```

Regardless of the defined upper and lower bounds of the array, MAXLOC will determine the lower bound index as '1'. Both MAXLOC and MINLOC index using positive integers. To find the actual index:

```
       INTEGER B(-100:100)
! Maxloc views the bounds as (1:201)
! If the largest element is located at index '-49'
       I = MAXLOC(B)
! Will return the index '52'
! To return the exact index for the largest element, insert:
       INDEX = LBOUND(B) - 1 + I
! Which is:  INDEX = (-100) - 1 + 52 = (-49)
       PRINT*, B(INDEX)
```

MAXVAL(ARRAY, DIM, MASK) or MAXVAL(ARRAY, MASK)

Returns the maximum value of the elements in the array along a dimension corresponding to the true elements of MASK.

ARRAY is an array of type integer or real.

DIM is an integer scalar in the range 1 ≤ DIM ≤ rank(ARRAY).

MASK (optional)
is an array or scalar of type logical that conforms to ARRAY in shape. If it is absent, the entire array is evaluated.

Class

Transformational function

Result Value

The result is an array of rank rank(ARRAY)-1, with the same data type as ARRAY. If DIM is missing or if ARRAY is of rank one, the result is a scalar.

If DIM is specified, each element of the result value contains the maximum value of all the elements that satisfy the condition specified by MASK along each vector of the dimension DIM. The array element subscripts in the result are (s_1, s_2, ..., $s_{(DIM-1)}$, $s_{(DIM+1)}$, ..., s_n), where n is the rank of ARRAY and DIM is the dimension specified by DIM.

If DIM is not specified, the function returns the maximum value of all applicable elements.

If ARRAY is zero-sized or the mask array has all .FALSE. values, the result value is the negative number of the largest magnitude, of the same type and kind type as ARRAY.

Because both DIM and MASK are optional, various combinations of arguments are possible. When the **-qintlog** option is specified with two arguments, the second argument refers to one of the following:

- MASK if it is an array of type integer, logical, byte or typeless
- DIM if it is a scalar of type integer, byte or typeless
- MASK if it is a scalar of type logical

Examples

```
! A is the array  | -41  33 25 |
!                 |  12 -61 11 |

! What is the largest value in the entire array?
      RES = MAXVAL(A)
! The result is 33

! What is the largest value in each column?
      RES = MAXVAL(A, DIM=1)
! The result is | 12 33 25 |

! What is the largest value in each row?
      RES = MAXVAL(A, DIM=2)
! The result is | 33 12 |

! What is the largest value in each row, considering only
! elements that are less than 30?
      RES = MAXVAL(A, DIM=2, MASK = A .LT. 30)
! The result is | 25 12 |
```

MERGE(TSOURCE, FSOURCE, MASK)

Selects between two values, or corresponding elements in two arrays. A logical mask determines whether to take each result element from the first or second argument.

TSOURCE is the source array to use when the corresponding element in the mask is true. It is an expression of any data type.

FSOURCE
: is the source array to use when the corresponding element in the mask is false. It must have the same data type and type parameters as tsource. It must conform in shape to tsource.

MASK
: is a logical expression that conforms to TSOURCE and FSOURCE in shape.

Class

Elemental function

Result Value

The result has the same shape and data type as TSOURCE and FSOURCE.

For each element in the result, the value of the corresponding element in MASK determines whether the value is taken from TSOURCE (if true) or FSOURCE (if false).

Examples

```
! TSOURCE is | A D G |, FSOURCE is | a d g |,
!            | B E H |             | b e h |
!            | C F I |             | c f i |
!
! and MASK is the array | T T T |
!                       | F F F |
!                       | F F F |

! Take the top row of TSOURCE, and the remaining elements
! from FSOURCE.
      RES = MERGE(TSOURCE, FSOURCE, MASK)
! The result is  | A D G |
!                | b e h |
!                | c f i |

! Evaluate IF (X .GT. Y) THEN
!             RES=6
!          ELSE
!             RES=12
!          END IF
! in a more concise form.
      RES = MERGE(6, 12, X .GT. Y)
```

MIN(A1, A2, A3, ...)

Minimum value.

- **A3, ...** are optional arguments. Any array that is itself an optional dummy argument must not be passed as an optional argument to this function unless it is present in the calling procedure.
- All the arguments must have the same type, either integer or real, and they all must have the same kind type parameter.

Class

Elemental function

Result Type and Attributes

Same as the arguments. (Some specific functions return results of a particular type.)

Result Value

The value of the result is that of the smallest argument.

Examples

MIN (-9.0, 7.0, 2.0) has the value -9.0.

If you evaluate MIN (10, 3, A), where A is an optional array argument in the calling procedure, PRESENT(A) must be true in the calling procedure.

Specific Name	Argument Type	Result Type	Pass As Arg?
AMIN0	any integer	default real	no
AMIN1	default real	default real	no
DMIN1	double precision real	double precision real	no
QMIN1	REAL(16)	REAL(16)	no
MIN0	any integer	same as argument	no
MIN1	any real	default integer	no

MINEXPONENT(X)

Returns the minimum (most negative) exponent in the model representing the numbers of the same type and kind type parameter as the argument.

X must be of type real. It may be scalar or array valued.

Class

Inquiry function

Result Type and Attributes

Default integer scalar.

Result Value

The result is the following:

```
   type          MINEXPONENT
----------       -----------
  real(4)           - 125
  real(8)           -1021
  real(16)           -968
```

Examples

```
MINEXPONENT(X) = -125 for X of type real(4).
```

See “Real Data Model” on page 522.

MINLOC(ARRAY, DIM, MASK) or MINLOC(ARRAY, MASK)

Locates the first element of an array along a dimension that has the minimum value of all elements corresponding to the true values of the mask. MINLOC will return the index referable to the position of the element using a positive integer.

ARRAY
is an array of type integer or real.

DIM
is a scalar integer in the range 1≤DIM≤*n*, where *n* is the rank of ARRAY.

MASK (optional)
is of type logical and conforms to ARRAY in shape. If it is absent, the default mask evaluation is `.TRUE.`; that is, the entire array is evaluated.

Class

Transformational function

Result Type and Attributes

If DIM is absent, the result is an integer array of rank one with a size equal to the rank of ARRAY. If DIM is present, the result is an integer array of rank rank(ARRAY)-1, and the shape is (s_1, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n), where n is the rank of ARRAY.

If there is no minimum value, perhaps because the array is zero-sized or the mask array has all `.FALSE.` values or there is no DIM argument, the return value is a zero-sized one-dimensional entity. If DIM is present, the result shape depends on the rank of ARRAY.

Result Value

The result indicates the subscript of the location of the minimum masked element of ARRAY. If more than one element is equal to this minimum value, the function finds the location of the first (in array element order). If DIM is specified, the result indicates the location of the minimum masked element along each vector of the dimension.

Because both DIM and MASK are optional, various combinations of arguments are possible. When the **-qintlog** option is specified with two arguments, the second argument refers to one of the following:

- MASK if it is an array of type integer, logical, byte or typeless
- DIM if it is a scalar of type integer, byte or typeless

- MASK if it is a scalar or type logical

The addition of the DIM argument modifies the behavior from XL Fortran Version 3.

Examples

```
! A is the array  |  4  9  8 -8 |
!                 |  2  1 -1  5 |
!                 |  9  4 -1  9 |
!                 | -7  5  7 -3 |

! Where is the smallest element of A?
       RES = MINLOC(A)
! The result is | 1 4 | because -8 is located at A(1,4).

! Where is the smallest element in each row of A that
! is not equal to -7?
       RES = MINLOC(A, DIM = 2, MASK = A .NE. -7)
! The result is | 4 3 3 4 | because these are the
! corresponding column locations of the smallest value
! in each row not equal ! to -7 (the values being
! -8,-1,-1,-3).
```

Regardless of the defined upper and lower bounds of the array, MINLOC will determine the lower bound index as '1'. Both MAXLOC and MINLOC index using positive integers. To find an actual index:

```
       INTEGER B(-100:100)
! Minloc views the bounds as (1:201)
! If the smallest element is located at index '-49'
       I = MINLOC(B)
! Will return the index '52'
! To return the exact index for the smallest element, insert:
       INDEX = LBOUND(B) - 1 + I
! Which is:  INDEX = (-100) - 1 + 52 = (-49)
       PRINT*, B(INDEX)
```

MINVAL(ARRAY, DIM, MASK) or MINVAL(ARRAY, MASK)

Returns the minimum value of the elements in the array along a dimension corresponding to the true elements of MASK.

ARRAY is an array of type integer or real.

DIM is an integer scalar in the range 1 ≤ DIM ≤ rank(ARRAY).

MASK (optional)
is an array or scalar of type logical that conforms to ARRAY in shape. If it is absent, the entire array is evaluated.

Class

Transformational function

Result Value

The result is an array of rank rank(ARRAY)-1, with the same data type as ARRAY. If DIM is missing or if ARRAY is of rank one, the result is a scalar.

If DIM is specified, each element of the result value contains the minimum value of all the elements that satisfy the condition specified by MASK along each vector of the dimension DIM. The array element subscripts in the result are (s_1, s_2, ..., $s_{(DIM-1)}$, $s_{(DIM+1)}$, ..., s_n), where n is the rank of ARRAY and DIM is the dimension specified by DIM.

If DIM is not specified, the function returns the minimum value of all applicable elements.

If ARRAY is zero-sized or the mask array has all .FALSE. values, the result value is the positive number of the largest magnitude, of the same type and kind type as ARRAY.

Because both DIM and MASK are optional, various combinations of arguments are possible. When the **-qintlog** option is specified with two arguments, the second argument refers to one of the following:

- MASK if it is an array of type integer, logical, byte or typeless
- DIM if it is a scalar of type integer, byte or typeless
- MASK if it is a scalar of type logical

Examples

```
! A is the array  | -41  33 25 |
!                 |  12 -61 11 |

! What is the smallest element in A?
       RES = MINVAL(A)
! The result is -61

! What is the smallest element in each column of A?
       RES = MINVAL(A, DIM=1)
! The result is | -41 -61 11 |

! What is the smallest element in each row of A?
       RES = MINVAL(A, DIM=2)
! The result is | -41 -61 |

! What is the smallest element in each row of A,
! considering only those elements that are
! greater than zero?
       RES = MINVAL(A, DIM=2, MASK = A .GT.0)
! The result is | 25 11 |
```

MOD(A, P)

Remainder function.

A must be of type integer or real.

P must be of the same type and kind type parameter as A.

Class

Elemental function

Result Type and Attributes

Same as A.

Result Value

- If P ≠ 0, the value of the result is A - INT(A/P) * P.
- If P = 0, the result is undefined.

Examples

MOD (3.0, 2.0) has the value 1.0.
MOD (8, 5) has the value 3.
MOD (-8, 5) has the value -3.
MOD (8, -5) has the value 3.
MOD (-8, -5) has the value -3.

Specific Name	Argument Type	Result Type	Pass As Arg?
MOD	any integer	same as argument	yes
AMOD	default real	default real	yes
DMOD	double precision real	double precision real	yes
QMOD	REAL(16)	REAL(16)	yes

MODULO(A, P)

Modulo function.

A must be of type integer or real.

P must be of the same type and kind type parameter as A.

Class

Elemental function

Result Type and Attributes

Same as A.

Result Value

- Case (i): A is of type integer. If P ≠ 0, MODULO (A, P) has the value R such that A = Q * P + R, where Q is an integer.

If $P > 0$, the inequalities $0 \le R < P$ hold.

If $P < 0$, $P < R \le 0$ hold.

If $P = 0$, the result is undefined.

- Case (ii): A is of type real. If $P \ne 0$, the value of the result is A - FLOOR (A / P) * P.

  If $P = 0$, the result is undefined.

Examples

MODULO (8, 5) has the value 3.
MODULO (-8, 5) has the value 2.
MODULO (8, -5) has the value -2.
MODULO (-8, -5) has the value -3.

MVBITS(FROM, FROMPOS, LEN, TO, TOPOS)

Copies a sequence of bits from one data object to another.

FROM must be of type integer. It is an INTENT(IN) argument.

FROMPOS must be of type integer and nonnegative. It is an INTENT(IN) argument. FROMPOS + LEN must be less than or equal to BIT_SIZE (FROM).

LEN must be of type integer and nonnegative. It is an INTENT(IN) argument.

TO must be a variable of type integer with the same kind type parameter value as FROM and may be the same variable as FROM. It is an INTENT(INOUT) argument. TO is set by copying the sequence of bits of length LEN, starting at position FROMPOS of FROM to position TOPOS of TO. No other bits of TO are altered. On return, the LEN bits of TO starting at TOPOS are equal to the value that the LEN bits of FROM starting at FROMPOS had on entry.

The bits are numbered 0 to BIT_SIZE(I)-1, from right to left.

TOPOS must be of type integer and nonnegative. It is an INTENT(IN) argument. TOPOS + LEN must be less than or equal to BIT_SIZE (TO).

Class

Elemental subroutine

Examples

If TO has the initial value 6, the value of TO is 5 after the statement

```
CALL MVBITS (7, 2, 2, TO, 0)
```

See “Integer Bit Model” on page 521.

NEAREST(X,S)

Returns the nearest different processor-representable number in the direction indicated by the sign of S (toward positive or negative infinity).

X must be of type real.

S must be of type real and not equal to zero.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

The result is the machine number different from and nearest to X in the direction of the infinity with the same sign as S.

Examples

NEAREST (3.0, 2.0) = 3.0 + $2.0^{(-22)}$. See "Real Data Model" on page 522.

NINT(A, KIND)

Nearest integer.

A must be of type real.

KIND (optional)
must be a scalar integer initialization expression.

Class

Elemental function

Result Type and Attributes

- Integer.
- If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of the default integer type.

Result Value

- If A > 0, NINT (A) has the value INT (A + 0.5).
- If A ≤ 0, NINT (A) has the value INT (A - 0.5).
- The result is undefined if its value cannot be represented in the specified integer type.

Examples

NINT (2.789) has the value 3. NINT (2.123) has the value 2.

Specific Name	Argument Type	Result Type	Pass As Arg?
NINT	default real	default integer	yes
IDNINT	double precision real	default integer	yes
IQNINT	REAL(16)	default integer	yes

NOT(I)

Performs a logical complement.

I must be of type integer.

Class

Elemental function

Result Type and Attributes

Same as I.

Result Value

The result has the value obtained by complementing I bit-by-bit according to the following table:

```
I NOT (I)
---------
1    0
0    1
```

The bits are numbered 0 to BIT_SIZE(I)-1, from right to left.

Examples

If I is represented by the string of bits `01010101`, NOT (I) has the string of bits `10101010`. See “Integer Bit Model” on page 521.

Specific Name	Argument Type	Result Type	Pass As Arg?
NOT	any integer	same as argument	yes

NULL(MOLD)

This function returns a pointer. The association status of the pointer is disassociated.

You must use the function without the **MOLD** argument in any of the following:

- initialization of an object in a declaration
- default initialization of a component
- in a **DATA** statement
- in a **STATIC** statement

You can use the function with or without the **MOLD** argument in any of the following:

- in the **PARAMETER** attribute
- on the right side of a pointer assignment
- in a structure constructor
- as an actual argument

MOLD (optional)
must be a pointer and can be of any type. The association status of the pointer can be undefined, disassociated, or associated. If the **MOLD** argument has an association status of associated, the target may be undefined.

Class

Transformational function.

Result Type and Attributes

If **MOLD** is present, the pointer's type, type parameter, and rank are the same as **MOLD**. If **MOLD** is not present, the pointer's type, type parameter and rank are determined as follows:

- same as the pointer that appears on the left hand side, for a pointer assignment
- same as the object, when initializing an object in a declaration
- same as the component, in a default initialization for a component
- same as the corresponding component, in a structure constructor
- same as the corresponding dummy argument, as an actual argument
- same as the corresponding pointer object, in a **DATA** statement
- same as the corresponding pointer object, in a **STATIC** statement

Result Value

The result is a pointer with disassociated association status.

Examples

```
! Using NULL() as an actual argument.
INTERFACE
   SUBROUTINE FOO(I, PR)
      INTEGER I
      REAL, POINTER:: PR
   END SUBROUTINE FOO
END INTERFACE

CALL FOO(5, NULL())
```

NUM_PARTHDS()

Returns the number of parallel Fortran threads the run time should create during execution of a program. This value is set by using the **PARTHDS** run-time option. If the user does not set the **PARTHDS** run-time option, the run time will set a default value for **PARTHDS**. In doing so, the run time may consider the following when setting the option:

- The number of processors on the machine
- The value specified in the run-time option **USRTHDS**.

Class

Inquiry function

Result Value

Default scalar integer

If the compiler option **-qsmp** has not been specified, then **NUM_PARTHDS** will always return a value of 1.

Examples

```
I = NUM_PARTHDS()
IF (I == 1) THEN
   CALL SINGLE_THREAD_ROUTINE()
ELSE
   CALL MULTI_THREAD_ROUTINE()
```

Specific Name	Argument Type	Result Type	Pass As Arg?
NUM_PARTHDS	default scalar integer	default scalar integer	no

Related Information

See the ″PARTHDS″ run-time option and the ″XLSMPOPTS″ run-time option in the *User's Guide*.

NUMBER_OF_PROCESSORS(DIM)

Returns a scalar of type default integer whose value is always 1.

DIM (optional)
must be a scalar integer and have a value of 1 (the rank of the processor array).

Class

System inquiry function

Result Type and Attributes

Default scalar integer which always has a value of 1 in a uniprocessor environment.

Examples

```
I = NUMBER_OF_PROCESSORS()       ! 1
J = NUMBER_OF_PROCESSORS(DIM=1)  ! 1
```

NUM_USRTHDS()

Returns the number of threads that will be explicitly created by the user during execution of the program. This value is set by using the **USRTHDS** run-time option.

Class

Inquiry function

Result Value

Default scalar integer

If the value has not been explicitly set using the **USRTHDS** run-time option, the default value is 0.

Specific Name	Argument Type	Result Type	Pass As Arg?
NUM_USRTHDS	default scalar integer	default scalar integer	no

Related Information

See the ″USRTHDS″ run-time option and the ″XLSMPOPTS″ run-time option in the *User's Guide*.

PACK(ARRAY, MASK, VECTOR)

Takes some or all elements from an array and packs them into a one-dimensional array, under the control of a mask.

ARRAY
is the source array, whose elements become part of the result. It can have any data type.

MASK
must be of type logical and must be conformable with ARRAY. It determines which elements are taken from the source array. If it is a scalar, its value applies to all elements in ARRAY.

VECTOR (optional)
is a padding array whose elements are used to fill out the result if there are not enough elements selected by the mask. It is a one-dimensional array that has the same data type as ARRAY and at least as many elements as there are true values in MASK. If MASK is a scalar with a value of .TRUE., VECTOR must have at least as many elements as there are array elements in ARRAY.

Class

Transformational function

Result Value

The result is always a one-dimensional array with the same data type as ARRAY.

The size of the result depends on the optional arguments:

- If VECTOR is specified, the size of the resultant array equals the size of VECTOR.
- Otherwise, it equals the number of true array elements in MASK, or the number of elements in ARRAY if MASK is a scalar with a value of .TRUE..

The array elements in ARRAY are taken in array element order to form the result. If the corresponding array element in MASK is .TRUE., the element from ARRAY is placed at the end of the result.

If any elements remain empty in the result (because VECTOR is present, and has more elements than there are .TRUE. values in mask), the remaining elements in the result are set to the corresponding values from VECTOR.

Examples

```
! A is the array  | 0  7  0 |
!                 | 1  0  3 |
!                 | 4  0  0 |

! Take only the non-zero elements of this sparse array.
! If there are less than six, fill in -1 for the rest.
RES = PACK(A, MASK= A .NE. 0, VECTOR=(/-1,-1,-1,-1,-1,-1/)
! The result is (/ 1, 4, 7, 3, -1, -1 /).

! Elements 1, 4, 7, and 3 are taken in order from A
! because the value of MASK is true only for these
! elements. The -1s are added to the result from VECTOR
! because the length (6) of VECTOR exceeds the number
! of .TRUE. values (4) in MASK.
```

PRECISION(X)

Returns the decimal precision in the model representing real numbers with the same kind type parameter as the argument.

X must be of type real or complex. It may be scalar or array valued.

Class

Inquiry function

Result Type and Attributes

Default integer scalar.

Result Value

The result is:

```
INT( (DIGITS(X) - 1) * LOG10(2) )
```

Therefore,

```
       Type                  Precision
--------------------      ---------
 real(4) , complex(4)          6
 real(8) , complex(8)         15
real(16) , complex(16)        31
```

Examples

PRECISION (X) = INT((24 - 1) * LOG10(2.)) = INT(6.92 ...) = 6 for X of type real(4). See "Real Data Model" on page 522.

PRESENT(A)

Determine whether an optional argument is present. If it is not present, you may only pass it as an optional argument to another procedure or pass it as an argument to PRESENT.

A is the name of an optional dummy argument that is accessible in the procedure in which the **PRESENT** function reference appears.

Class

Inquiry function

Result Type and Attributes

Default logical scalar.

Result Value

The result is .TRUE. if the actual argument is present (that is, if it was passed to the current procedure in the specified dummy argument), and .FALSE. otherwise.

Examples

```
      SUBROUTINE SUB (X, Y)
        REAL, OPTIONAL :: Y
        IF (PRESENT (Y)) THEN
! In this section, we can use y like any other variable.
           X = X + Y
           PRINT *, SQRT(Y)
        ELSE
! In this section, we cannot define or reference y.
           X = X + 5
! We can pass it to another procedure, but only if
! sub2 declares the corresponding argument as optional.
           CALL SUB2 (Z, Y)
        ENDIF
      END SUBROUTINE SUB
```

Related Information

"OPTIONAL" on page 361

PROCESSORS_SHAPE()

Returns a zero-sized array.

Class

System inquiry function

Result Type and Attributes

Default integer array of rank one, whose size is equal to the rank of the processor array. In a uniprocessor environment, the result is a zero-sized vector.

Result Value

The value of the result is the shape of the processor array.

Examples

```
I=PROCESSORS_SHAPE()
! Zero-sized vector of type default integer
```

PRODUCT(ARRAY, DIM, MASK) or PRODUCT(ARRAY, MASK)

Multiplies together all elements in an entire array, or selected elements from all vectors along a dimension.

ARRAY is an array with a numeric data type.

DIM is an integer scalar in the range 1 ≤ DIM ≤ rank(ARRAY).

MASK (optional)
is a logical expression that conforms with ARRAY in shape. If MASK is a scalar, the scalar value applies to all elements in ARRAY.

Class

Transformational function

Result Value

If DIM is present, the result is an array of rank rank(ARRAY)-1 and the same data type as ARRAY. If DIM is missing, or if MASK has a rank of one, the result is a scalar.

The result is calculated by one of the following methods:

Method 1:
If only ARRAY is specified, the result is the product of all its array elements. If ARRAY is a zero-sized array, the result is equal to one.

Method 2:
: If ARRAY and MASK are both specified, the result is the product of those array elements of ARRAY that have a corresponding true array element in MASK. If MASK has no elements with a value of .TRUE., the result is equal to one.

Method 3:
: If DIM is also specified, the result value equals the product of the array elements of ARRAY along dimension DIM that have a corresponding true array element in MASK.

Because both DIM and MASK are optional, various combinations of arguments are possible. When the **-qintlog** option is specified with two arguments, the second argument refers to one of the following:

- MASK if it is an array of type integer, logical, byte or typeless
- DIM if it is a scalar of type integer, byte or typeless
- MASK if it is a scalar of type logical

Examples

Method 1:

```
! Multiply all elements in an array.
       RES = PRODUCT( (/2, 3, 4/) )
! The result is 24 because (2 * 3 * 4) = 24.

! Do the same for a two-dimensional array.
       RES = PRODUCT( (/2, 3, 4/), (/4, 5, 6/) )
! The result is 2880.  All elements are multiplied.
```

Method 2:

```
! A is the array (/ -3, -7, -5, 2, 3 /)
! Multiply all elements of the array that are > -5.
       RES = PRODUCT(A, MASK = A .GT. -5)
! The result is -18 because (-3 * 2 * 3) = -18.
```

Method 3:

```
! A is the array | -2  5  7 |
!                |  3 -4  3 |
! Find the product of each column in A.
       RES = PRODUCT(A, DIM = 1)
! The result is | -6 -20 21 | because (-2 * 3) =  -6
!                                     ( 5 * -4 ) = -20
!                                     ( 7 *  3 ) = 21

! Find the product of each row in A.
       RES = PRODUCT(A, DIM = 2)
! The result is | -70 -36 |
! because (-2 *  5 * 7) = -70
!         (3 * -4 * 3) = -36

! Find the product of each row in A, considering
! only those elements greater than zero.
```

```
       RES = PRODUCT(A, DIM = 2, MASK = A .GT. 0)
! The result is | 35 9 | because ( 5 * 7) = 35
!                                 (3 * 3) =  9
```

QCMPLX(X, Y)

Convert to extended complex type.

X must be of type integer, real, or complex.

Y (optional) must be of type integer or real. It must not be present if X is of type complex.

Class

Elemental function

Result Type and Attributes

It is of type extended complex.

Result Value

- If Y is absent and X is not complex, it is as if Y were present with the value of zero.
- If Y is absent and X is complex, it is as if Y were present with the value AIMAG(X) and X were present with the value REAL(X).
- QCMPLX(X, Y) has the complex value whose real part is REAL(X, KIND=16) and whose imaginary part is REAL(Y, KIND=16).

Examples

QCMPLX (-3) has the value (-3.0Q0, 0.0Q0).

Specific Name	Argument Type	Result Type	Pass As Arg?
QCMPLX	REAL(16)	COMPLEX(16)	no

Related Information

“CMPLX(X, Y, KIND)” on page 539, “DCMPLX(X, Y)” on page 550.

QEXT(A)

Convert to extended precision real type.

A must be of type integer, or real.

Class

Elemental function

Result Type and Attributes

Extended precision real.

Result Value

- If A is of type extended precision real, QEXT(A) = A.
- If A is of type integer or real, the result is the exact extended precision representation of A.

Examples

QEXT (-3) has the value -3.0Q0.

Specific Name	Argument Type	Result Type	Pass As Arg?
QFLOAT	any integer	REAL(16)	no
QEXT	default real	REAL(16)	no
QEXTD	double precision real	REAL(16)	no

RADIX(X)

Returns the base of the model representing numbers of the same type and kind type parameter as the argument.

X must be of type integer or real. It may be scalar or array valued.

Class

Inquiry function

Result Type and Attributes

Default integer scalar.

Result Value

The result is the base of the model representing numbers of the same kind and type as X. The result is always 2. See the models under "Data Representation Models" on page 521.

RAND()

Not recommended. Generates uniform random numbers, positive real numbers greater than or equal to 0.0 and less than 1.0. Instead, use the RANDOM_NUMBER intrinsic subroutine.

Class

None (does not correspond to any of the defined categories).

Result Type and Attributes

real(4) scalar.

Related Information

"SRAND(SEED)" on page 626 can be used to specify a seed value for the random number sequence.

If the function result is assigned to an array, all array elements receive the same value.

Examples

The following is an example of a program using the RAND function.

```
DO I = 1, 5
   R = RAND()
   PRINT *, R
ENDDO
END
```

The following is sample output generated by the above program:

```
0.2251586914
0.8285522461
0.6456298828
0.2496948242
0.2215576172
```

This function only has a specific name.

RANDOM_NUMBER(HARVEST)

Returns one pseudo-random number or an array of pseudo-random numbers from the uniform distribution over the range $0 \le x < 1$.

HARVEST
must be of type real. It is an INTENT(OUT) argument. It may be a scalar or array variable. It is set to pseudo-random numbers from the uniform distribution in the interval $0 \le x < 1$.

Class

Subroutine

Examples

```
REAL X, Y (10, 10)
! Initialize X with a pseudo-random number
CALL RANDOM_NUMBER (HARVEST = X)
CALL RANDOM_NUMBER (Y)
! X and Y contain uniformly distributed random numbers
```

RANDOM_SEED(SIZE, PUT, GET, GENERATOR)

Restarts or queries the pseudo-random number generator used by RANDOM_NUMBER.

There must either be exactly one or no arguments present.

SIZE (optional)
must be scalar and of type default integer. It is an INTENT(OUT) argument. It is set to the number of default

type integers (N) that are needed to hold the value of the seed, which is an 8-byte variable.

PUT (optional)
must be a default integer array of rank one and size ≥ N. It is an INTENT(IN) argument. The seed for the current generator is transferred from it.

GET (optional)
must be a default integer array of rank one and size ≥ N. It is an INTENT(OUT) argument. The seed for the current generator is transferred to it.

GENERATOR (optional)
must be a scalar and of type default integer. It is an INTENT(IN) argument. Its value determines the random number generator to be used subsequently. The value must be either 1 or 2.

Random_seed allows the user to toggle between two random number generators. Generator 1 is the default. Each generator maintains a private seed and normally resumes its cycle after the last number it generated.

Generator 1 uses the multiplicative congruential method, with

```
S(I+1) = ( 16807.0 * S(I) ) mod (2.0**31-1)
```

and

```
X(I+1) = S(I+1) / (2.0**31-1)
```

Generator 1 cycles after 2**31-2 random numbers.

Generator 2 also uses the multiplicative congruential method, with

```
S(I+1) = ( 44,485,709,377,909.0 * S(I) )
              mod (2.0**48)
```

and

```
X(I+1) = S(I+1) / (2.0**48)
```

Generator 2 cycles after (2**46)-1 random numbers. Although generator 1 is the default (for reasons of backwards compatibility) the use of generator 2 is recommended for new programs since it typically runs faster than generator 1 and has a longer period.

If no argument is present, the seed of the current generator is set to the default value 1d0.

Class

Subroutine

Examples

```
CALL RANDOM_SEED
   ! Current generator sets its seed to 1d0
CALL RANDOM_SEED (SIZE = K)
   ! Sets K = 64 / BIT_SIZE( 0 )
CALL RANDOM_SEED (PUT = SEED (1 : K))
   ! Transfer seed to current generator
CALL RANDOM_SEED (GET = OLD (1 : K))
   ! Transfer seed from current generator
```

RANGE(X)

Returns the decimal exponent range in the model representing integer or real numbers with the same kind type parameter as the argument.

X must be of type integer, real, or complex. It may be scalar or array valued.

Class

Inquiry function

Result Type and Attributes

Default integer scalar.

Result Value

1. For an integer argument, the result is:

   `INT( LOG10( HUGE(X) ) )`

2. For a real or complex argument, the result is:

   `INT( MIN( LOG10( HUGE(X) ), -LOG10( TINY(X) ) ) )`

Thus:

```
Type                        RANGE
------------------------------
integer(1)                    2
integer(2)                    4
integer(4)                    9
integer(8)                   18
real(4) , complex(4)         37
real(8) , complex(8)        307
real(16) , complex(16)      291
```

Examples

X is of type real(4):
HUGE(X) = 0.34E+39
TINY(X) = 0.11E-37
RANGE(X) = 37

See "Data Representation Models" on page 521.

REAL(A, KIND)

Convert to real type.

A must be of type integer, real, or complex.

KIND (optional)
must be a scalar integer initialization expression.

Class

Elemental function

Result Type and Attributes

- Real.
- Case (i): If A is of type integer or real and KIND is present, the kind type parameter is that specified by KIND. If A is of type integer or real and KIND is not present, the kind type parameter is the kind type parameter of the default real type.
- Case (ii): If A is of type complex and KIND is present, the kind type parameter is that specified by KIND. If A is of type complex and KIND is not present, the kind type parameter is the kind type parameter of A.

Result Value

- Case (i): If A is of type integer or real, the result is equal to a kind-dependent approximation to A.
- Case (ii): If A is of type complex, the result is equal to a kind-dependent approximation to the real part of A.

Examples

REAL (-3) has the value -3.0. REAL ((3.2, 2.1)) has the value 3.2.

Specific Name	Argument Type	Result Type	Pass As Arg?
REAL	default integer	default real	no
FLOAT	any integer 1	default real	no
SNGL	double precision real	default real	no
SNGLQ	REAL(16)	default real	no
DREAL	double complex	double precision real	no
QREAL	COMPLEX(16)	REAL(16)	no

Notes:

1. The extension is the ability to specify a nondefault integer argument.

REPEAT(STRING, NCOPIES)

Concatenate several copies of a string.

STRING must be scalar and of type character.

NCOPIES must be scalar and of type integer. Its value must not be negative.

Class

Transformational function

Result Type and Attributes

Character scalar with a length equal to NCOPIES * LENGTH(STRING), with the same kind type parameter as STRING.

Result Value

The value of the result is the concatenation of NCOPIES copies of STRING.

Examples

REPEAT ('H', 2) has the value 'HH'. REPEAT ('XYZ', 0) has the value of a zero-length string.

RESHAPE(SOURCE, SHAPE, PAD, ORDER)

Constructs an array of a specified shape from the elements of a given array.

SOURCE is an array of any type, which supplies the elements for the result array.

SHAPE defines the shape of the result array. It is an integer array of up to 20 elements, with rank one and of a constant size. All elements are either positive integers or zero.

PAD (optional)
is used to fill in extra values if SOURCE is reshaped into a larger array. It is an array of the same data type as SOURCE. If it is absent or is a zero-sized array, you can only make SOURCE into another array of the same size or smaller.

ORDER (optional)
is an integer array of rank one with a constant size. Its elements must be a permutation of (1, 2, ..., SIZE(SHAPE)). You can use it to insert elements in the result in an order of dimensions other than the normal (1, 2, ..., rank(RESULT)).

Class

Transformational function

Result Value

The result is an array with shape SHAPE. It has the same data type as SOURCE.

The array elements of SOURCE are placed into the result in the order of dimensions as specified by ORDER, or in the usual order for array elements if ORDER is not specified.

The array elements of SOURCE are followed by the array elements of PAD in array element order, and followed by additional copies of PAD until all of the elements of the result are set.

Examples

```
! Turn a rank-1 array into a 3x4 array of the
! same size.
RES= RESHAPE( (/A,B,C,D,E,F,G,H,I,J,K,L/), (/3,4/)
! The result is | A D G J |
!               | B E H K |
!               | C F I L |

! Turn a rank-1 array into a larger 3x5 array.
! Keep repeating -1 and -2 values for any
! elements not filled by the source array.
! Fill the rows first, then the columns.
RES= RESHAPE( (/1,2,3,4,5,6/), (/3,5/), &
  (/-1,-2/), (/2,1/) )
! The result is | 1  2  3  4  5  |
!               |  6 -1 -2 -1 -2 |
!               | -1 -2 -1 -2 -1 |
```

Related Information

"SHAPE(SOURCE)" on page 618.

RRSPACING(X)

Returns the reciprocal of the relative spacing of the model numbers near the argument value.

X must be of type real.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

The result is:

ABS(FRACTION(X)) * FLOAT(RADIX(X))$^{\mathrm{DIGITS(X)}}$

Examples

RRSPACING (-3.0) = 0.75 * 2^{24}. See "Real Data Model" on page 522.

RSHIFT(I, SHIFT)

Performs a logical shift to the right.

I
must be of type integer.

SHIFT
must be of type integer. It must be non-negative and less than or equal to BIT_SIZE(I).

Class

Elemental function

Result Type and Attributes

Same as I.

Result Value

- The result has the value obtained by shifting the bits of I by SHIFT positions to the right.
- Vacated bits are filled with the sign bit.
- The bits are numbered 0 to BIT_SIZE(I)-1, from right to left.

Examples

RSHIFT (3, 1) has the result 1.

RSHIFT (3, 2) has the result 0.

Specific Name	Argument Type	Result Type	Pass As Arg?
RSHIFT	any integer	same as argument	yes

SCALE(X,I)

Returns the scaled value: `X * 2.0`^{I}

X
must be of type real.

I
must be of type integer.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

The result is determined from the following:

```
X * 2.0^I
```

SCALE (X, I) = X * (2.0^I)

Examples

SCALE (4.0, 3) = 4.0 * (2^3) = 32.0. See "Real Data Model" on page 522.

SCAN(STRING, SET, BACK)

Scan a string for any one of the characters in a set of characters.

STRING must be of type character.

SET must be of type character with the same kind type parameter as STRING.

BACK (optional)
must be of type logical.

Class

Elemental function

Result Type and Attributes

Default integer.

Result Value

- Case (i): If BACK is absent or is present with the value .FALSE. and if STRING contains at least one character that is in SET, the value of the result is the position of the leftmost character of STRING that is in SET.
- Case (ii): If BACK is present with the value .TRUE. and if STRING contains at least one character that is in SET, the value of the result is the position of the rightmost character of STRING that is in SET.
- Case (iii): The value of the result is zero if no character of STRING is in SET or if the length of STRING or SET is zero.

Examples

- Case (i): SCAN ('FORTRAN', 'TR') has the value 3.
- Case (ii): SCAN ('FORTRAN', 'TR', BACK = .TRUE.) has the value 5.
- Case (iii): SCAN ('FORTRAN', 'BCD') has the value 0.

SELECTED_INT_KIND(R)

Returns a value of the kind type parameter of an integer data type that represents all integer values n with $-10^R < n < 10^R$.

R must be a scalar of type integer.

Class

Transformational function

Result Type and Attributes

Default integer scalar.

Result Value

- The result has a value equal to the value of the kind type parameter of an integer data type that represents all values n in the range values n with $-10^R < n < 10^R$, or if no such kind type parameter is available, the result is -1.
- If more than one kind type parameter meets the criteria, the value returned is the one with the smallest decimal exponent range.

Examples

SELECTED_INT_KIND (9) has the value 4, signifying that an INTEGER with kind type 4 can represent all values from 10^{-9} to 10^9.

Related Information

Kind type parameters supported by XL Fortran are defined in "Type Parameters and Specifiers" on page 27.

SELECTED_REAL_KIND(P, R)

Returns a value of the kind type parameter of a real data type with decimal precision of at least P digits and a decimal exponent range of at least R.

P (optional) must be scalar and of type integer.

R (optional) must be scalar and of type integer.

Class

Transformational function

Result Type and Attributes

Default integer scalar.

Result Value

- The result has a value equal to a value of the kind type parameter of a real data type with decimal precision, as returned by the function PRECISION, of at least P digits and a decimal exponent range, as returned by the function RANGE, of at least R, or if no such kind type parameter is available,
  - If the precision is not available, the result is -1.
  - If the exponent range is not available, the result is -2.
  - If neither is available, the result is -3.

- If more than one kind type parameter value meets the criteria, the value returned is the one with the smallest decimal precision, unless there are several such values, in which case the smallest of these kind values is returned.

Examples

SELECTED_REAL_KIND (6, 70) has the value 8.

Related Information

Kind type parameters supported by XL Fortran are defined in "Type Parameters and Specifiers" on page 27.

SET_EXPONENT(X,I)

Returns the number whose fractional part is the fractional part of the model representation of X, and whose exponent part is I.

X must be of type real.

I must be of type integer.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

- If X = 0 the result is zero.
- Otherwise, the result is:

  $FRACTION(X) * 2.0^{I}$

Examples

SET_EXPONENT (10.5, 1) = $0.65625 * 2.0^{1} = 1.3125$

See "Real Data Model" on page 522.

SHAPE(SOURCE)

Returns the shape of an array or scalar.

SOURCE is an array or scalar of any data type. It must not be a disassociated pointer, allocatable array that is not allocated, or assumed-size array.

Class

Inquiry function

Result Value

The result is a one-dimensional default integer array whose elements define the shape of SOURCES. The extent of each dimension in SOURCES is returned in the corresponding element in the result array.

Related Information

"RESHAPE(SOURCE, SHAPE, PAD, ORDER)" on page 613.

Examples

```
! A is the array  | 7 6 3 1 |
!                 | 2 4 0 9 |
!                 | 5 7 6 8 |
!
       RES = SHAPE( A )
! The result is | 3  4 | because A is a rank-2 array
! with 3 elements in each column and 4 elements in
! each row.
```

SIGN(A, B)

Returns the absolute value of A times the sign of B. If A is non-zero, you can use the result to determine whether B is negative or non-negative, as the sign of the result is the same as the sign of B.

Note that if you have declared B as **REAL(4)** or **REAL(8)**, and B has a negative zero value, the sign of the result depends on whether you have specified the **-qxlf90=signedzero** compiler option.

A must be of type integer or real.

B must be of the same type and kind type parameter as A.

Class

Elemental function

Result Type and Attributes

Same as A.

Result Value

The result is *sgn**|A|, where:

- *sgn* = -1, if either of the following is true:
  - B < 0
  - B is a **REAL(4)** or **REAL(8)** number with a value of negative 0, and you have specified the **-qxlf90=signedzero** option
- *sgn* = 1, otherwise.

Fortran 95 allows a processor to distinguish between a positive and a negative real zero, whereas Fortran 90 did not. Using the **-qxlf90=signedzero** option allows you to specify the Fortran 95 behavior (except in the case of **REAL(16)**

numbers), which is consistent with the IEEE standard for binary floating-point arithmetic. **-qxlf90=signedzero** is the default for the **xlf95**, **xlf95_r**, and **xlf95_r7** invocation commands.

Examples

SIGN (-3.0, 2.0) has the value 3.0.

Specific Name	Argument Type	Result Type	Pass As Arg?
SIGN	default real	default real	yes
ISIGN	any integer **1**	same as argument	yes
DSIGN	double precision real	double precision real	yes
QSIGN	REAL(16)	REAL(16)	yes

Notes:

1. The extension is the ability to specify a nondefault integer argument.

Related Information

See the "-qxlf90 Option" in the *User's Guide*.

SIGNAL(I, PROC)

The SIGNAL procedure allows a program to specify a procedure to be invoked upon receipt of a specific operating-system signal.

I
: is an integer that specifies the value of the signal to be acted upon. It is an INTENT(IN) argument. Available signal values are defined in the C include file **signal.h**; a subset of signal values is defined in the Fortran include file **fexcp.h**.

PROC
: specifies the user-defined procedure to be invoked when the process receives the specified signal (I). It is an INTENT(IN) argument.

Class

Subroutine

Examples

```
      INCLUDE 'fexcp.h'
      INTEGER   SIGUSR1
      EXTERNAL  USRINT
! Set exception handler to produce the traceback code.
! The SIGTRAP is defined in the include file fexcp.h.
! xl__trce is a procedure in the XL Fortran
! run-time library.  It generates the traceback code.
      CALL SIGNAL(SIGTRAP, XL__TRCE)
      ...
```

```
! Use user-defined procedure USRINT to handle the signal
! SIGUSR1.
      CALL SIGNAL(SIGUSR1, USRINT)
      ...
```

Related Information

See the **signal** subroutine in the *Technical Reference: Base Operating System and Extensions Volume 1* for details about the underlying implementation.

The "-qsigtrap Option" in the *User's Guide* allows you to set a handler for **SIGTRAP** signals through a compiler option.

SIN(X)

Sine function.

X must be of type real or complex. If X is real, it is regarded as a value in radians. If X is complex, its real part is regarded as a value in radians.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

It approximates sin(X).

Examples

SIN (1.0) has the value 0.84147098 (approximately).

Specific Name	Argument Type	Result Type	Pass As Arg?
SIN	default real	default real	yes
DSIN	double precision real	double precision real	yes
QSIN	REAL(16)	REAL(16)	yes
CSIN 1	default complex	default complex	yes
CDSIN 2	double complex	double complex	yes
ZSIN 2	double complex	double complex	yes
CQSIN 2	COMPLEX(16)	COMPLEX(16)	yes

Given that X is a complex number in the form a + bi, where i = $(-1)^{½}$:

1. abs(b) must be less than or equal to 88.7228; a is any real value.
2. abs(b) must be less than or equal to 709.7827; a is any real value.

SIND(X)

Sine function. Argument in degrees.

X must be of type real.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

- It approximates sin(X), where X has a value in degrees.

Examples

SIND (90.0) has the value 1.0.

Specific Name	Argument Type	Result Type	Pass As Arg?
SIND	default real	default real	yes
DSIND	double precision real	double precision real	yes
QSIND	REAL(16)	REAL(16)	yes

SINH(X)

Hyperbolic sine function.

X must be of type real.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

The result has a value equal to sinh(x).

Examples

SINH (1.0) has the value 1.1752012 (approximately).

Specific Name	Argument Type	Result Type	Pass As Arg?
SINH 1	default real	default real	yes
DSINH 2	double precision real	double precision real	yes
QSINH 2	REAL(16)	REAL(16)	yes

Notes:

1. abs(X) must be less than or equal to 89.4159.
2. abs(X) must be less than or equal to 709.7827.

SIZE(ARRAY, DIM)

Returns the extent of an array along a specified dimension or the total number of elements in the array.

ARRAY is an array of any data type. It must not be a scalar, disassociated pointer, or allocatable array that is not allocated. It can be an assumed-size array if DIM is present and has a value that is less than the rank of ARRAY.

DIM (optional)
is an integer scalar in the range 1 ≤ DIM ≤ rank(ARRAY).

Class

Inquiry function

Result Type and Attributes

Default integer scalar.

Result Value

The result equals the extent of ARRAY along dimension DIM; or, if DIM is not specified, it is the total number of array elements in ARRAY.

Examples

```
! A is the array  | 1 -4  7 -10 |
!                 | 2  5 -8  11 |
!                 | 3  6  9 -12 |

       RES = SIZE( A )
! The result is 12 because there are 12 elements in A.

       RES = SIZE( A, DIM = 1)
! The result is 3 because there are 3 rows in A.

       RES = SIZE( A, DIM = 2)
! The result is 4 because there are 4 columns in A.
```

SPACING(X)

Returns the absolute spacing of the model numbers near the argument value.

X must be of type real.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

If X is not 0, the result is:

$2.0^{EXPONENT(X) - DIGITS(X)}$

If X is 0, the result is the same as that of TINY(X).

Examples

SPACING (3.0) = $2.0^{2 - 24} = 2.0^{(-22)}$ See "Real Data Model" on page 522.

SPREAD(SOURCE, DIM, NCOPIES)

Replicates an array in an additional dimension by making copies of existing elements along that dimension.

SOURCE
can be an array or scalar. It can have any data type. The rank of SOURCE has a maximum value of 19.

DIM
is an integer scalar in the range 1 ≤ DIM ≤ rank(SOURCE)+1. Unlike most other array intrinsic functions, **SPREAD** requires the DIM argument.

NCOPIES
is an integer scalar. It becomes the extent of the extra dimension added to the result.

Class

Transformational function

Result Type and Attributes

The result is an array of rank rank(SOURCE)+1 and with the same type and type parameters as source.

Result Value

If SOURCE is a scalar, the result is a one-dimensional array with NCOPIES elements, each with value SOURCE.

If SOURCE is an array, the result is an array of rank rank(SOURCE) + 1. Along dimension DIM, each array element of the result is equal to the corresponding array element in SOURCE.

If NCOPIES is less than or equal to zero, the result is a zero-sized array.

Examples

```
! A is the array (/ -4.7, 6.1, 0.3 /)

      RES = SPREAD( A, DIM = 1, NCOPIES = 3 )
! The result is    | -4.7 6.1 0.3 |
!                  | -4.7 6.1 0.3 |
!                  | -4.7 6.1 0.3 |
! DIM=1 extends each column.  Each element in RES(:,1)
! becomes a copy of A(1), each element in RES(:,2) becomes
! a copy of A(2), and so on.

      RES = SPREAD( A, DIM = 2, NCOPIES = 3 )
! The result is    | -4.7 -4.7 -4.7 |
!                  |  6.1  6.1  6.1 |
!                  |  0.3  0.3  0.3 |
! DIM=2 extends each row.  Each element in RES(1,:)
! becomes a copy of A(1), each element in RES(2,:)
! becomes a copy of A(2), and so on.

      RES = SPREAD( A, DIM = 2, NCOPIES = 0 )
! The result is (/ /) (a zero-sized array).
```

SQRT(X)

Square root.

X must be of type real or complex. Unless X is complex, its value must be greater than or equal to zero.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

- It has a value equal to the square root of X.
- If the result type is complex, its value is the principal value with the real part greater than or equal to zero. If the real part is zero, the imaginary part is greater than or equal to zero.

Examples

SQRT (4.0) has the value 2.0.

Specific Name	Argument Type	Result Type	Pass As Arg?
SQRT	default real	default real	yes
DSQRT	double precision real	double precision real	yes
QSQRT	REAL(16)	REAL(16)	yes

Specific Name	Argument Type	Result Type	Pass As Arg?
CSQRT 1	default complex	default complex	yes
CDSQRT 1	double complex	double complex	yes
ZSQRT 1	COMPLEX(8)	COMPLEX(8)	yes
CQSQRT 1	COMPLEX(16)	COMPLEX(16)	yes

Given that X is a complex number in the form a + bi, where i = $(-1)^{½}$:

1. abs(X) + abs(a) must be less than or equal to 1.797693 * 10^{308}.

SRAND(SEED)

Provides the seed value used by the random number generator function RAND.

SEED must be scalar. It must be of type **REAL(4)** when used to provide a seed value for the **RAND** function, or of type **INTEGER(4)** when used to provide a seed value for the **IRAND** service and utility function. It is an INTENT(IN) argument.

Class

Subroutine

Examples

The following is an example of a program using the SRAND subroutine.

```
CALL SRAND(0.5)
DO I = 1, 5
   R = RAND()
   PRINT *,R
ENDDO
END
```

The following is sample output generated by the above program:

```
0.3984375000
0.4048461914
0.1644897461
0.1281738281E-01
0.2313232422E-01
```

SUM(ARRAY, DIM, MASK) or SUM(ARRAY, MASK)

Calculates the sum of selected elements in an array.

ARRAY is an array of numeric type, whose elements you want to sum.

DIM is an integer scalar in the range 1 ≤ DIM ≤ rank(ARRAY).

MASK (optional)
is a logical expression. If it is an array, it must conform with ARRAY in shape. If MASK is a scalar, the scalar value applies to all elements in ARRAY.

Class

Transformational function

Result Value

If DIM is present, the result is an array of rank rank(ARRAY)-1, with the same data type as ARRAY. If DIM is missing, or if MASK has a rank of one, the result is a scalar.

The result is calculated by one of the following methods:

Method 1:
If only ARRAY is specified, the result equals the sum of all the array elements of ARRAY. If ARRAY is a zero-sized array, the result equals zero.

Method 2:
If ARRAY and MASK are both specified, the result equals the sum of the array elements of ARRAY that have a corresponding array element in MASK with a value of .TRUE.. If MASK has no elements with a value of .TRUE., the result is equal to zero.

Method 3:
If DIM is also specified, the result value equals the sum of the array elements of ARRAY along dimension DIM that have a corresponding true array element in MASK.

Because both DIM and MASK are optional, various combinations of arguments are possible. When the **-qintlog** option is specified with two arguments, the second argument refers to one of the following:

- MASK if it is an array of type integer, logical, byte or typeless
- DIM if it is a scalar of type integer, byte or typeless
- MASK if it is a scalar of type logical

Examples

Method 1:

```
! Sum all the elements in an array.
       RES = SUM( (/2, 3, 4 /) )
! The result is 9 because (2+3+4) = 9
```

Method 2:

```
! A is the array (/ -3, -7, -5, 2, 3 /)
! Sum all elements that are greater than -5.
      RES = SUM( A, MASK = A .GT. -5 )
! The result is 2 because (-3 + 2 + 3) = 2
```

Method 3:

```
! B is the array | 4 2 3 |
!                | 7 8 5 |

! Sum the elements in each column.
      RES = SUM(B, DIM = 1)
! The result is | 11 10 8 | because (4 + 7) = 11
!                                   (2 + 8) = 10
!                                   (3 + 5) =  8

! Sum the elements in each row.
      RES = SUM(B, DIM = 2)
! The result is | 9 20 | because (4 + 2 + 3) =  9
!                                (7 + 8 + 5) = 20

! Sum the elements in each row, considering only
! those elements greater than two.
      RES = SUM(B, DIM = 2, MASK = B .GT. 2)
! The result is | 7 20 | because (4 + 3) =  7
!                                (7 + 8 + 5) = 20
```

SYSTEM(CMD, RESULT)

Passes a command to the operating system for execution. The current process pauses until the command is completed and control is returned from the operating system. An added, optional argument to the subroutine will allow recovery of any return code information from the operating system.

CMD
must be scalar and of type character, specifying the command to execute and any command-line arguments. It is an INTENT(IN) argument.

RESULT
must be a scalar variable of type INTEGER(4). If the argument is not an INTEGER(4) variable, the compiler will generate an (S) level error message. It is an optional INTENT(OUT) argument. The format of the information returned in RESULT is the same as the format returned from the WAIT system call.

Class

Subroutine

Examples

```
INTEGER        ULIMIT
CHARACTER(32)  CMD
...
```

```
! Check the system ulimit.
      CMD = 'ulimit > ./fort.99'
      CALL SYSTEM(CMD)
      READ(99, *) ULIMIT
      IF (ULIMIT .LT. 2097151) THEN
         ...
     INTEGER RC
     RC=99
     CALL SYSTEM("/bin/test 1 -EQ 2",RC)
     IF (IAND(RC,'ff'z) .EQ. 0) then
       RC = IAND( ISHFT(RC,-8), 'ff'z )
     ELSE
       RC = -1
     ENDIF
```

Related Information

See the **system** subroutine in the *Technical Reference: Base Operating System and Extensions Volume 1* for details about the underlying implementation.

SYSTEM_CLOCK(COUNT, COUNT_RATE, COUNT_MAX)

Returns integer data from a real-time clock.

COUNT (optional)
must be scalar and of type default integer. It is an INTENT(OUT) argument. It is set to a value based on the current value of the processor clock. The value is incremented by one for each clock count until the value COUNT_MAX is reached and is reset to zero at the next count. It lies in the range of 0 to COUNT_MAX if there is a clock.

COUNT_RATE (optional)
must be scalar and of type default integer. It is an INTENT(OUT) argument. It is set to the number of processor clock counts per second, or to zero if there is no clock.

COUNT_MAX (optional)
must be scalar and of type default integer. It is an INTENT(OUT) argument. It is set to the maximum value that COUNT can have, or to zero if there is no clock.

Class

Subroutine

Examples

```
! The following example shows how to interpret values
! returned by the subroutine SYSTEM_CLOCK. The processor
! clock is a 24-hour clock. After the call to SYSTEM_CLOCK,
! the COUNT contains the day time expressed in clock ticks
! per second. The number of ticks per second is available
```

```
! in the COUNT_RATE. The COUNT_RATE value is processor-
! dependent.

INTEGER, DIMENSION(8) :: IV
TIME_SYNC: DO
CALL DATE_AND_TIME(VALUES=IV)
IHR  = IV(5)
IMIN = IV(6)
ISEC = IV(7)
CALL SYSTEM_CLOCK(COUNT=IC, COUNT_RATE=IR, COUNT_MAX=IM)
CALL DATE_AND_TIME(VALUES=IV)

IF ((IHR == IV(5)) .AND. (IMIN == IV(6)) .AND. &
  (ISEC == IV(7))) EXIT TIME_SYNC

END DO TIME_SYNC

IDAY_SEC = 3600*IHR + IMIN*60 + ISEC
IDAY_TICKS = IDAY_SEC * IR

IF (IDAY_TICKS /= IC) THEN
  STOP 'clock error'
ENDIF
END
```

TAN(X)

Tangent function.

X must be of type real.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

The result approximates tan(X), where X has a value in radians.

Examples

TAN (1.0) has the value 1.5574077 (approximately).

Specific Name	Argument Type	Result Type	Pass As Arg?
TAN	default real	default real	yes
DTAN	double precision real	double precision real	yes
QTAN	REAL(16)	REAL(16)	yes

TAND(X)

Tangent function. Argument in degrees.

X must be of type real.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

The result approximates tan(X), where X has a value in degrees.

Examples

TAND (45.0) has the value 1.0.

Specific Name	Argument Type	Result Type	Pass As Arg?
TAND	default real	default real	yes
DTAND	double precision real	double precision real	yes
QTAND	REAL(16)	REAL(16)	yes

TANH(X)

Hyperbolic tangent function.

X must be of type real.

Class

Elemental function

Result Type and Attributes

Same as X.

Result Value

The result has a value equal to tanh(X).

Examples

TANH (1.0) has the value 0.76159416 (approximately).

Specific Name	Argument Type	Result Type	Pass As Arg?
TANH	default real	default real	yes
DTANH	double precision real	double precision real	yes
QTANH	REAL(16)	REAL(16)	yes

TINY(X)

Returns the smallest positive number in the model representing numbers of the same type and kind type parameter as the argument.

X must be of type real. It may be scalar or array valued.

Class

Inquiry function

Result Type and Attributes

Scalar with the same type as X.

Result Value

The result is:

`2.0`$^{(MINEXPONENT(X)-1)}$ `for real X`

Examples

TINY (X) = float(2)$^{(-126)}$ = 1.17549351e-38. See "Real Data Model" on page 522.

TRANSFER(SOURCE, MOLD, SIZE)

Returns a result with a physical representation identical to that of SOURCE but interpreted with the type and type parameters of MOLD.

It performs a low-level conversion between types without any sign extension, rounding, blank padding, or other alteration that may occur using other methods of conversion.

SOURCE is the data entity whose bitwise value you want to transfer to a different type. It may be of any type, and may be scalar or array valued.

MOLD is a data entity that has the type characteristics you want for the result. It may be of any type, and may be scalar or array valued. Its value is not used, only its type characteristics.

SIZE (optional)
is the number of elements for the output result. It must be a scalar integer. The corresponding actual argument must not be an optional dummy argument.

Class

Transformational function

Result Type and Attributes

The same type and type parameters as MOLD.

If MOLD is a scalar and SIZE is absent, the result is a scalar.

If MOLD is array valued and SIZE is absent, the result is array valued and of rank one, with the smallest size that is physically large enough to hold SOURCE.

If SIZE is present, the result is array valued of rank one and size SIZE.

Result Value

The physical representation of the result is the same as SOURCE, truncated if the result is smaller or with an undefined trailing portion if the result is larger.

Because the physical representation is unchanged, it is possible to undo the results of TRANSFER as long as the result is not truncated:

```
      REAL(4) X = 3.141
      DOUBLE PRECISION I, J(6) = (/1,2,3,4,5,6/)

! Because x is transferred to a larger representation
! and then back, its value is unchanged.
      X = TRANSFER( TRANSFER( X, I ), X )

! j is transferred into a real(4) array large enough to
! hold all its elements, then back into an array of
! its original size, so its value is unchanged too.
      J = TRANSFER( TRANSFER( J, X ), J, SIZE=SIZE(J) )
```

Examples

TRANSFER (1082130432, 0.0) is 4.0.

TRANSFER ((/1.1,2.2,3.3/), (/(0.0,0.0)/)) is a complex rank-one array of length two whose first element has the value (1.1, 2.2) and whose second element has a real part with the value 3.3. The imaginary part of the second element is undefined.

TRANSFER ((/1.1,2.2,3.3/), (/(0.0,0.0)/), 1) has the value (/(1.1,2.2)/).

TRANSPOSE(MATRIX)

Transposes a two-dimensional array, turning each column into a row and each row into a column.

MATRIX is an array of any data type, with a rank of two.

Class

Transformational function

Result Value

The result is a two-dimensional array of the same data type as MATRIX.

The shape of the result is (n,m) where the shape of MATRIX is (m,n). For example, if the shape of MATRIX is (2,3), the shape of the result is (3,2).

Each element (i,j) in the result has the value MATRIX (j,i) for i in the range 1-n and j in the range 1-m.

Examples

```
! A is the array  | 0 -5  8 -7 |
!                 | 2  4 -1  1 |
!                 | 7  5  6 -6 |
! Transpose the columns and rows of A.
      RES = TRANSPOSE( A )
! The result is   |  0  2  7 |
!                 | -5  4  5 |
!                 |  8 -1  6 |
!                 | -7  1 -6 |
```

TRIM(STRING)

Returns the argument with trailing blank characters removed.

STRING must be of type character and must be a scalar.

Class

Transformational function

Result Type and Attributes

Character with the same kind type parameter value as STRING and with a length that is the length of STRING less the number of trailing blanks in STRING.

Result Value

- The value of the result is the same as STRING, except trailing blanks are removed.
- If STRING contains no nonblank characters, the result has zero length.

Examples

TRIM ('ƀAƀBƀƀ') has the value 'ƀAƀB'.

UBOUND(ARRAY, DIM)

Returns the upper bounds of each dimension in an array, or the upper bound of a specified dimension.

ARRAY is the array whose upper bounds you want to determine. Its bounds must be defined: that is, it must not be a disassociated pointer or an allocatable array that is not allocated, and if its size is assumed, you can only examine one dimension.

DIM (optional)
is an integer scalar in the range 1 ≤ DIM ≤ rank(ARRAY). The corresponding actual argument must not be an optional dummy argument.

Class

Inquiry function

Result Type and Attributes

Default integer.

If DIM is present, the result is a scalar. If it is not present, the result is a one-dimensional array with one element for each dimension in ARRAY.

Result Value

Each element in the result corresponds to a dimension of ARRAY. If ARRAY is a whole array or array structure component, these values are equal to the upper bounds. If ARRAY is an array section or expression that is not a whole array or array structure component, the values represent the number of elements in each dimension, which may be different than the declared upper bounds of the original array. If a dimension is zero-sized, the corresponding element in the result is zero, regardless of the value of the upper bound.

Examples

```
! This array illustrates the way UBOUND works with
! different ranges for dimensions.
       REAL A(1:10, -4:5, 4:-5)

       RES=UBOUND( A )
! The result is (/ 10, 5, 0 /).

       RES=UBOUND( A(:,:,:) )
! The result is (/ 10, 10, 0 /) because the argument
! is an array section.

       RES=UBOUND( A(4:10,-4:1,:) )
! The result is (/ 7, 6, 0 /), because for an array section,
! it is the number of elements that is significant.
```

UNPACK(VECTOR, MASK, FIELD)

Takes some or all elements from a one-dimensional array and rearranges them into another, possibly larger, array.

VECTOR
is a one-dimensional array of any data type. There must be at least as many elements in VECTOR as there are .TRUE. values in MASK.

MASK
is a logical array that determines where the elements of VECTOR are placed when they are unpacked.

FIELD must have the same shape as the mask argument, and the same data type as VECTOR. Its elements are inserted into the result array wherever the corresponding MASK element has the value .FALSE..

Class

Transformational function

Result Value

The result is an array with the same shape as MASK and the same data type as VECTOR.

The elements of the result are filled in array-element order: if the corresponding element in MASK is .TRUE., the result element is filled by the next element of VECTOR; otherwise, it is filled by the corresponding element of FIELD.

Examples

```
! VECTOR is the array (/ 5, 6, 7, 8 /),
! MASK is | F T T |, FIELD is | -1 -4 -7 |
!         | T F F |           | -2 -5 -8 |
!         | F F T |           | -3 -6 -9 |

! Turn the one-dimensional vector into a two-dimensional
! array.  The elements of VECTOR are placed into the .TRUE.
! positions in MASK, and the remaining elements are
! made up of negative values from FIELD.
      RES = UNPACK( VECTOR, MASK, FIELD )
! The result is | -1  6  7 |
!               |  5 -5 -8 |
!               | -3 -6  8 |

! Do the same transformation, but using all zeros for the
! replacement values of FIELD.
       RES = UNPACK( VECTOR, MASK, FIELD = 0 )
! The result is | 0 6 7 |
!               | 5 0 0 |
!               | 0 0 8 |
```

VERIFY(STRING, SET, BACK)

Verify that a set of characters contains all the characters in a string by identifying the position of the first character in a string of characters that does not appear in a given set of characters.

STRING must be of type character.

SET must be of type character with the same kind type parameter as STRING.

BACK (optional)
must be of type logical.

Class

Elemental function

Result Type and Attributes

Default integer.

Result Value

- Case (i): If BACK is absent or present with the value .FALSE. and if STRING contains at least one character that is not in SET, the value of the result is the position of the leftmost character of STRING that is not in SET.
- Case (ii): If BACK is present with the value .TRUE. and if STRING contains at least one character that is not in SET, the value of the result is the position of the rightmost character of STRING that is not in SET.
- Case (iii): The value of the result is zero if each character in STRING is in SET or if STRING has zero length.

Examples

- Case (i): VERIFY ('ABBA', 'A') has the value 2.
- Case (ii): VERIFY ('ABBA', 'A', BACK = .TRUE.) has the value 3.
- Case (iii): VERIFY ('ABBA', 'AB') has the value 0.

Chapter 13. Service and Utility Procedures

XL Fortran provides utility services that are available to the Fortran programmer. There are two categories of service and utility procedures: efficient floating–point control and inquiry procedures, and general service and utility procedures. The rules for these two categories of procedures differ from one another. This chapter describes the general rules for the two categories of procedures, then provides an alphabetical reference to the procedures.

Efficient Floating-point Control and Inquiry Procedures

XL Fortran provides several procedures that allow you to query and control the floating-point status and control register of the processor directly. These procedures are more efficient than the fpgets and fpsets subroutines because they are mapped into inlined machine instructions that manipulate the floating-point status and control register (fpscr) directly.

XL Fortran supplies the module `xlf_fp_util`, which contains the interfaces and data type definitions for these procedures and the definitions for the named constants that are needed by the procedures. This module enables type checking of these procedures at compile time rather than at link time. You can use the argument names listed in the examples as the names for keyword arguments when calling the procedure. The following files are supplied for the modules `xlf_fp_util`:

File names	File type	Locations
xlf_fp_util.mod	module symbol file (32–bit)	• /usr/lpp/xlf/include_32_d10 • /usr/lpp/xlf/include_32_d7 **Note:** The files in these directories are exact copies of one another.
	module symbol file (64–bit)	/usr/lpp/xlf/include_64

To use these procedures, you must add a `USE XLF_FP_UTIL` statement to your source file. For more information on **USE**, see “USE” on page 408.

If there are name conflicts (for example if the accessing subprogram has an entity with the same name as a module entity), use the **ONLY** clause or the renaming features of the **USE** statement. For example,

```
USE XLF_FP_UTIL, NULL1 => get_fpscr, NULL2 => set_fpscr
```

© Copyright IBM Corp. 1990, 2000

When compiling with the **-U** option, you must code the names of these procedures in all lowercase. We will show the names in lowercase here as a reminder.

The fpscr procedures are:

- clr_fpscr_flags
- fp_trap
- get_fpscr
- get_fpscr_flags
- get_round_mode
- set_fpscr
- set_fpscr_flags
- set_round_mode

The following table lists the constants that are used with the fpscr procedures:

Family	Constant	**Description**
General	FPSCR_KIND	The kind type parameter for a fpscr flags variable
	FP_MODE_KIND	The kind type parameter for fp_trap arguments and results
IEEE Rounding Modes	FP_RND_RN	Round toward nearest (default)
	FP_RND_RZ	Round toward zero
	FP_RND_RP	Round toward plus infinity
	FP_RND_RM	Round toward minus infinity
	FP_RND_MODE	Used to obtain the rounding mode from a fpscr flags variable or value
IEEE Exception Enable Flags 1	TRP_INEXACT	Enable inexact trap
	TRP_DIV_BY_ZERO	Enable divide-by-zero trap
	TRP_UNDERFLOW	Enable underflow trap
	TRP_OVERFLOW	Enable overflow trap
	TRP_INVALID	Enable invalid trap
	FP_ENBL_SUMM	Trap enable summary or enable all
IEEE Exception Status Flags	FP_INVALID	Invalid operation exception
	FP_OVERFLOW	Overflow exception
	FP_UNDERFLOW	Underflow exception
	FP_DIV_BY_ZERO	Divide-by-zero exception
	FP_INEXACT	Inexact exception
	FP_ALL_IEEE_XCP	All IEEE exceptions summary flags
	FP_COMMON_IEEE_XCP	All IEEE exceptions summary flags excluding the FP_INEXACT exception
Machine Specific Exception Details Flags	FP_INV_SNAN	Signalling NaN
	FP_INV_ISI	Infinity – Infinity
	FP_INV_IDI	Infinity / Infinity
	FP_INV_ZDZ	0 / 0
	FP_INV_IMZ	Infinity * 0
	FP_INV_CMP	Unordered compare
	FP_INV_SQRT	Square root of negative number
	FP_INV_CVI	Conversion to integer error
	FP_INV_VXSOFT	Software request

Machine Specific Exception Summary Flags	FP_ANY_XCP	Any exception summary flag
	FP_ALL_XCP	All exceptions summary flags
	FP_COMMON_XCP	All exceptions summary flags excluding the FP_INEXACT exception
fp_trap constants **2**	FP_TRAP_SYNC	Precise trapping on
	FP_TRAP_OFF	Trapping off
	FP_TRAP_QUERY	Query trapping mode
	FP_TRAP_IMP	Non-recoverable imprecise trapping on
	FP_TRAP_IMP_REC	Recoverable imprecise trapping on
	FP_TRAP_FASTMODE	Select fastest available mode
	FP_TRAP_ERROR	Error condition
	FP_TRAP_UNIMPL	Requested mode not available

Notes:

1. In order to enable exception trapping, you must set the desired IEEE Exception Enable Flags and,
   - change the mode of the user process to allow floating-point exceptions to generate traps with a call to fp_trap, or,
   - compile your program with the appropriate **–qflttrap** suboption. For more information on the **–qflttrap** compiler option and its suboptions, see the *User's Guide*.
2. For more information on fp_trap constants, see fp_trap in *Technical Reference: Base Operating System and Extensions Volume 1*.

General Service and Utility Procedures

All of the procedures described in this chapter that do not belong to the `xlf_fp_util` module belong in this category. To ensure that the functions are given the correct type and that naming conflicts are avoided, use these procedures in one of the following two ways:

1. XL Fortran supplies the module `xlfutility`, which contains the interfaces and data type definitions for these procedures (and the derived-type definitions required for the **dtime_**, **etime_**, **idate_**, and **itime_** procedures). XL Fortran flags arguments that are not compatible with the interface specification in type, kind, and rank. These modules enable type checking of these procedures at compile time rather than at link time. The argument names in the module interface are taken from the examples defined below. The following files are supplied for the modules `xlfutility` and `xlfutility_extname`:

File names	File type	Locations
• xlfutility.f • xlfutility_extname.f	source file	/usr/lpp/xlf/samples/modules
• xlfutility.mod • xlfutility_extname.mod	module symbol file (32–bit)	• /usr/lpp/xlf/include_32_d10 • /usr/lpp/xlf/include_32_d7 **Note:** The files in these directories are exact copies of one another.
	module symbol file (64–bit)	/usr/lpp/xlf/include_64

You can use the precompiled module by adding a **USE** statement to your source file (see "USE" on page 408 for details). As well, you can modify the module source file and recompile it to suit your needs. Use the `xlfutility_extname` files for procedures compiled with the **-qextname** option. The source file `xlfutility_extname.f` has no underscores following procedure names, while `xlfutility.f` includes underscores for some procedures names (as listed in this chapter).

If there are name conflicts (for example if the accessing subprogram has an entity with the same name as a module entity), use the **ONLY** clause or the renaming features of the **USE** statement. For example,

```
USE XLFUTILITY, NULL1 => DTIME_, NULL2 => ETIME_
```

2. Because these procedures are not intrinsic procedures:
   - You must declare their type to avoid potential problems with implicit typing.
   - When compiling with the **-U** option, you must code the names of these procedures in all lowercase to match the names in the XL Fortran libraries. We will show the names in lowercase here as a reminder.

   To avoid conflicts with names in the **libc** library, some procedure names end with an underscore. When coding calls to these procedures, you can:
   - Instead of entering the underscore, use the **-brename** linker option to change the name at link time:

     ```
     xlf -brename:flush,flush_ calls_flush.f
     ```

     This method works best if you need to rename only a small number of procedures.
   - Instead of typing the underscore, use the **-qextname** compiler option to add it to the end of each name:

     ```
     xlf -qextname calls_flush.f
     ```

This method is recommended for programs already written without the underscore following the routine name. The XL Fortran library contains additional entry points, such as **fpgets_**, so that calls to procedures that do not use trailing underscores still resolve with **-qextname**.

- Depending on the way your program is structured and the particular libraries and object files it uses, you may have difficulty using **-qextname** or **-brename**. In this case, enter the underscores after the appropriate names in the source file:

```
PRINT *, IRTC()  ! No underscore in this name
CALL FLUSH_(10)  ! But there is one in this name
```

If your program calls the following procedures, there are restrictions on the common block and external procedure names that you can use:

XLF-Provided Function Name	Common Block or External Procedure Name You Cannot Use
mclock	times
rand	irand

Note: The **mvbits** subroutine that was in XL Fortran Version 2 is now an intrinsic subroutine, "MVBITS(FROM, FROMPOS, LEN, TO, TOPOS)" on page 597.

List of Service and Utility Procedures

The following table lists the service and utility procedures in the `XLFUTILITY` and `XLF_FP_UTIL` modules.

Note: CHARACTER(n) means that you can specify any length for the variable.

Table 16. Service and Utility Procedures

Procedures

alarm_

The **alarm_** function sends an alarm signal at time TIME to invoke the function FUNC. The returned value is the remaining time from the last alarm.

Format/ Example

```
INTEGER(4) REMAINING, TIME,
  FUNC, alarm_
REMAINING =  alarm_ (TIME, FUNC)
```

Table 16. Service and Utility Procedures (continued)

Procedures

bic_

The **bic_** subroutine sets bit X1 of X2 to 0. X1 has a value range 0 ≤ X1 ≤ 31.

Format/ Example

```
INTEGER(4) X1, X2
CALL bic_ (X1, X2)
```

bis_

The **bis_** subroutine sets bit X1 of X2 to 1. X1 has a value range 0 ≤ X1 ≤ 31.

Format/ Example

```
INTEGER(4) X1, X2
CALL bis_ (X1, X2)
```

bit_

The **bit_** function returns the value **.TRUE.** in BITCHK if bit X1 of X2 equals 1. Otherwise, **bit_** returns the value 0. X1 has a value range 0 ≤ X1 ≤ 31.

Format/ Example

```
INTEGER(4) X2, X1
LOGICAL BITCHK, bit_
BITCHK = bit_ (X1, X2)
```

clock_

The **clock_** function returns the time in hh:mm:ss format. This function is different from the operating system clock function.

Format/ Example

```
CHARACTER(8) C, clock_
C = clock_()
```

Table 16. Service and Utility Procedures (continued)

Procedures

clr_fpscr_flags

The clr_fpscr_flags subroutine clears the floating-point status and control register flags you specify in the MASK argument. Flags that you do not specify in MASK remain unaffected. MASK must be of type INTEGER(FPSCR_KIND). You can manipulate the MASK using the intrinsic procedures described in "Integer Bit Model" on page 521.

For more information on the FPSCR constants, see "fpscr constants" on page 641.

Format/ Example

```
USE XLF_FP_UTIL
INTEGER(FPSCR_KIND) MASK

MASK=(IOR(FP_OVERFLOW,FP_UNDERFLOW))
CALL clr_fpscr_flags(MASK)
```

For another example of the clr_fpscr_flags subroutine, see "get_fpscr_flags" on page 653.

ctime_

The **ctime_** subroutine converts the system time TIME to a 26-character ASCII string.

Format/ Example

```
INTEGER(4) TIME
CHARACTER(26) STR
CALL ctime_(STR, TIME)
```

date

The date function returns the current date in mm/dd/yy format.

Format/ Example

```
CHARACTER(8) D, date
D = date()
```

Table 16. Service and Utility Procedures (continued)

Procedures

---

dtime_

The **dtime_** function sets the time accounting information for the user time and system time in DTIME_STRUCT. The returned value, DELTA, is the sum of the user time and the system time since the last call to **dtime_**. The resolution for all timing is 1/100 of a second. The output appears in units of seconds.

Format/ Example

```
REAL(4) DELTA, dtime_
TYPE TB_TYPE
  SEQUENCE
  REAL(4) USRTIME
  REAL(4) SYSTIME
END TYPE
TYPE (TB_TYPE) DTIME_STRUCT
DELTA = dtime_(DTIME_STRUCT)
```

---

etime_

The **etime_** function sets the user-elapsed time and system-elapsed time in ETIME_STRUCT since the start of the execution of a process. The returned value, ELAPSED, is the sum of the user-elapsed time and the system-elapsed time. The resolution for all timing is 1/100 of a second. The output appears in units of seconds.

Format/ Example

```
REAL(4) ELAPSED, etime_
TYPE TB_TYPE
  SEQUENCE
  REAL(4) USRTIME
  REAL(4) SYSTIME
END TYPE
TYPE (TB_TYPE) ETIME_STRUCT
ELAPSED = etime_(ETIME_STRUCT)
```

---

exit_

The **exit_** subroutine stops execution of the process with the exit status of EXIT_STATUS.

Format/ Example

```
INTEGER(4) EXIT_STATUS
CALL exit_(EXIT_STATUS)
```

Table 16. Service and Utility Procedures (continued)

Procedures

fdate_
 	
The **fdate_** subroutine returns the date and time in a 26-character ASCII string. In the example, the date and time are returned in STR.
 	
Format/ Example
 	
`CHARACTER(26) STR`
`CALL fdate_(STR)`

Procedures

fiosetup_

The **fiosetup_** function sets up the requested I/O behavior for the logical unit specified by UNIT. The request is specified by argument COMMAND. The argument ARGUMENT is an argument to the COMMAND. The Fortran include file 'fiosetup_.h' is supplied with the compiler to define symbolic constants for the fiosetup_ arguments and error return codes.

UNIT
is a logical unit that is currently connected to a file

COMMAND
IO_CMD_FLUSH_AFTER_WRITE (1). Specifies whether the buffers of the specified UNIT be flushed after every WRITE statement.

IO_CMD_FLUSH_BEFORE_READ (2). Specifies whether the buffers of the specified UNIT be flushed before every READ statement. This can be used to refresh the data currently in the buffers.

ARGUMENT
IO_ARG_FLUSH_YES (1). Causes the buffers of the specified UNIT to be flushed after every WRITE statement. This argument should be specified with the commands IO_CMD_FLUSH_AFTER_WRITE and IO_CMD_FLUSH_BEFORE_READ.

IO_ARG_FLUSH_NO (0) Instructs the I/O library to flush buffers at its own discretion. Note the units connected to certain device types must be flushed after each WRITE operation regardless of the IO_CMD_FLUSH_AFTER_WRITE setting. Such devices include terminals and pipes. This argument should be specified with the commands IO_CMD_FLUSH_AFTER_WRITE and IO_CMD_FLUSH_BEFORE_READ. This is the default setting for both commands.

Format/ Example

```
FUNCTION fiosetup_
(UNIT, COMMAND, ARGUMENT)
INTEGER(4) fiosetup_, IRESULT
INTEGER(4) UNIT, COMMAND, ARGUMENT
INCLUDE 'fiosetup_.h'
OPEN ( UNIT=42, FILE="foo", ...)
IRESULT = fiosetup_(42, &
     IO_CMD_FLUSH_AFTER_WRITE, &
     IO_ARG_FLUSH_YES)
```

```
The service routine FIOSETUP_ returns 0 if it succeeds. Otherwise, it returns one of the
following error codes:
```

IO_ERR_NO_RTE (1000)
the run-time environment is not running.
IO_ERR_BAD_UNIT(1001)
the specified UNIT is unconnected.
IO_ERR_BAD_CMD (1002)
invalid command.
IO_ERR_BAD_ARG (1003)
invalid argument.

Table 16. Service and Utility Procedures (continued)

Procedures

flush_
 	
The **flush_** subroutine flushes the contents of the input/output buffer for the logical unit LUNIT. The value of LUNIT must be within the range 0 ≤ LUNIT ≤ 2**31-1.
 	
Format/ Example
 	
`INTEGER(4) LUNIT`
`CALL flush_(LUNIT)`

fpgets fpsets
 	
The subroutines **fpgets** and **fpsets** retrieve and set the status of the floating-point operations, respectively. The include file fpdc.h contains the data declarations (specification statements) for the two subroutines. The include file fpdt.h contains the data initializations (data statements) and must be included in a block data program unit.
 	
fpgets retrieves the floating-point process status and stores the result in a logical array called fpstat.
 	
fpsets sets the floating-point status equal to the logical array fpstat.
 	
This array contains logical values that can be used to specify floating-point rounding modes. See the "FPGETS and FPSETS Subroutines" in the *User's Guide* for examples and information on the elements of the fpstat array.
Note: The `XLF_FP_UTIL` module provides procedures for manipulating the status of floating-point operations that are more efficient than the fpgets and fpsets subroutines. For more information, see "Efficient Floating-point Control and Inquiry Procedures" on page 639.
 	
Format/ Example
 	
`CALL fpgets( fpstat )`
`...`
`CALL fpsets( fpstat )`
`BLOCK DATA`
`INCLUDE '/usr/include/fpdc.h'`
`INCLUDE '/usr/include/fpdt.h'`
`END`

Procedures

fp_trap

The fp_trap function allows you to query or change the mode of the user process to allow floating-point exceptions to generate traps. fp_trap returns an INTEGER(FP_MODE_KIND) in the form of an fp_trap constant. The argument TRAP_MODE should be an fp_trap constant. For information on fp_trap constants, see "fpscr constants" on page 641.

For more information, see fp_trap in *Technical Reference: Base Operating System and Extensions Volume 1*.

Format/ Example

```
USE XLF_FP_UTIL
INTEGER(FP_MODE_KIND) FP_MODE, TRAP_MODE

TRAP_MODE = FP_TRAP_IMP
FP_MODE = fp_trap(TRAP_MODE)
```

For another example of how to use fp_trap, see "set_fpscr_flags" on page 663.

---

ftell_ ftell64_

The **ftell_** function returns the offset of the current byte relative to the beginning of the file associated with the specified logical unit UNIT. If the unit is not connected, the **ftell_** function returns -1.

The **ftell64_** function is identical to the **ftell_** function with the exception that it can operate on files larger than 2 gigabytes in large file enabled file systems.

The offset returned by the **ftell_** or **ftell64_** function is the result of previously completed I/O operations. No references to **ftell_** or **ftell64_** on a unit with outstanding asynchronous data transfer operations are allowed until the matching **WAIT** statements for all outstanding asynchronous data transfer operations on the same unit are executed.

The offset returned by the **ftell_** or **ftell64_** function is the absolute offset of the current byte relative to the beginning of the file. This means that all bytes from the beginning of the file to the current byte are counted, including the data of the records and record terminators if they are present.

Format/ Example

```
INTEGER(4) ftell_, UNIT1, UNIT2, IRESULT
INTEGER(8) ftell64_, IRESULT8

UNIT1 = 42
IRESULT = ftell_(UNIT1)

UNIT2 = 44
IRESULT8 = ftell64_(UNIT2)
! Unit 44 might be connected to a
! file larger than 2 gigabytes
```

Table 16. Service and Utility Procedures (continued)

Procedures

getarg

The **getarg** subroutine returns a command line argument of the current process. I1 is an integer argument that specifies which command line argument to return. C1 is an argument of character type and will contain, upon return from **getarg**, the command line argument. If I1 is equal to 0, the program name is returned.

Format/ Example

```
INTEGER(4) I1
CHARACTER(n) C1
CALL getarg(I1,C1)
```

getcwd_

The **getcwd_** function retrieves the pathname NAME of the current working directory where the maximum length is 1024 characters. The returned value, IS_CWD, is 0 if successful, and an error number otherwise.

Format/ Example

```
INTEGER(4) IS_CWD, getcwd_
CHARACTER(1024) NAME
IS_CWD = getcwd_ (NAME)
```

getfd

Given a Fortran logical unit, the **getfd** function returns the underlying file descriptor for that unit, or -1 if the unit is not connected.
Note: Because XL Fortran does its own I/O buffering, using this function may require special care, as described in "Mixed-Language Input and Output" in the *User's Guide*.

Format/ Example

```
INTEGER(4) LUNIT, FD, getfd
FD = getfd(LUNIT)
```

Table 16. Service and Utility Procedures (continued)

Procedures

get_fpscr

The get_fpscr function returns the current value of the floating-point status and control register (fpscr) of the processor.

Format/ Example

```
USE XLF_FP_UTIL
INTEGER(FPSCR_KIND) FPSCR

FPSCR=get_fpscr()
```

get_fpscr_flags

The get_fpscr_flags function returns the current state of the floating-point status and control register flags you specify in the MASK argument. MASK must be of type INTEGER(FPSCR_KIND). You can manipulate the MASK using the intrinsics described in "Integer Bit Model" on page 521.

For more information on the FPSCR constants, see "fpscr constants" on page 641.

Format/ Example

```
USE XLF_FP_UTIL

! ...

IF (get_fpscr_flags(IOR(FP_DIV_BY_ZERO,FP_INVALID)) .NE. 0) THEN
  ! Either Divide-by-zero or an invalid operation occurred.

  ! ...

  ! After processing the exception, the exception flags are
  ! cleared.
  CALL clr_fpscr_flags(IOR(FP_DIV_BY_ZERO,FP_INVALID))
END IF
```

getgid_

The **getgid_** function returns the group id of a process, where GROUP_ID is the requested real group id of the calling process.

Format/ Example

```
INTEGER(4) GROUP_ID, getgid_
GROUP_ID = getgid_()
```

Table 16. Service and Utility Procedures (continued)

Procedures

getlog_
 	
The **getlog_** subroutine stores the user's login name in NAME. NAME has a maximum length of 8 characters. If the user's login name is not found, NAME is filled with blanks.
 	
Format/ Example
 	
`CHARACTER(8) NAME`
`CALL getlog_ (NAME)`

getpid_
 	
The **getpid_** function returns the process id of the current process in PROCESS_ID.
 	
Format/ Example
 	
`INTEGER(4) PROCESS_ID, getpid_`
`PROCESS_ID = getpid_()`

get_round_mode
 	
The get_round_mode function returns the current floating-point rounding mode. The return value will be one of the constants FP_RND_RN, FP_RND_RZ, FP_RND_RP or FP_RND_RM. For more information on the rounding mode constants, see "fpscr constants" on page 641.
 	
Format/ Example
 	
`USE XLF_FP_UTIL`
`INTEGER(FPSCR_KIND) MODE`
 	
`MODE=get_round_mode()`
`IF (MODE .EQ. FP_RND_RZ) THEN`
`! ...`
`END IF`

getuid_
 	
The **getuid_** function returns the real user id of the current process in USER_ID.
 	
Format/ Example
 	
`INTEGER(4) USER_ID, getuid_`
`USER_ID = getuid_()`

Table 16. Service and Utility Procedures (continued)

Procedures

global_timef

The **global_timef** function returns the elapsed time in milliseconds since the first call to **global_timef** was first executed among all running threads. The first call to global_timef returns 0.0 in milliseconds. This function returns the global timing results from all running threads. For thread-specific timing results, see the "timef" function on page 664.

Format/ Example

```
INTEGER N
REAL(8) global_timef, T1, T2, T3
  T1 = global_timef()   ! returns 0.0
    DO I = 1, N         ! loop 1
      H = I + 1000
    END DO
    DO I = 1, N         ! loop 2
      M = I + 2000
    END DO
  T2 = global_timef()
! returns the elapsed time of
!  loop 1 and loop 2
    DO I = 1, N         ! loop 3
      M = I + 3000
    END DO
  T3 = global_timef()
! returns the elapsed time of
!  loop 1, 2 and 3
END
```

gmtime_

The **gmtime_** subroutine converts the system time STIME into the array TARRAY. The data is stored in TARRAY in the following order:

seconds (0 to 59)
minutes (0 to 59)
hours (0 to 23)
day of the month (1 to 31)
month of the year (0 to 11)
year (year = current year - 1900)
day of week (Sunday = 0)
day of year (0 to 365)
daylight saving time (0 or 1)

Format/ Example

```
INTEGER(4) STIME, TARRAY(9)
CALL gmtime_(STIME, TARRAY)
```

Table 16. Service and Utility Procedures (continued)

Procedures

hostnm_

The **hostnm_** function retrieves the machine's host name NAME. NAME has a maximum length of 32 characters. The returned value, ISHOST, is 0 if the host name is found, and an error number otherwise.

Format/ Example

```
INTEGER(4) ISHOST, hostnm_
CHARACTER(32) NAME
ISHOST = hostnm_ (NAME)
```

---

iargc

The **iargc** function returns an integer that represents the number of arguments following the program name that have been entered on the command line at run time.

Format/ Example

```
INTEGER(4) I1, iargc
I1 = iargc()
```

---

idate_

The **idate_** subroutine returns the current date in a numerical format containing the day, month and year in IDATE_STRUCT.

Format/ Example

```
TYPE IDATE_TYPE
  SEQUENCE
  INTEGER(4) IDAY
  INTEGER(4) IMONTH
  INTEGER(4) IYEAR
END TYPE
TYPE (IDATE_TYPE) IDATE_STRUCT
CALL idate_(IDATE_STRUCT)
```

---

ierrno_

The **ierrno_** function returns the error number, SYSERROR, of the last detected system error.

Format/ Example

```
INTEGER(4) SYSERROR, ierrno_
SYSERROR = ierrno_()
```

Table 16. Service and Utility Procedures (continued)

Procedures

irand

The **irand** function generates a positive integer number greater than 0 and less than or equal to 32768. The intrinsic subroutine "SRAND(SEED)" on page 626 is used to provide the seed value for the random number generator.

Format/ Example

```
INTEGER(4) I1, irand
CALL SRAND(I1)
I1 = irand()
```

---

irtc

The **irtc** function returns an INTEGER(8) value of the number of nanoseconds since the initial value of the machine's real-time clock.

Format/ Example

```
INTEGER(8) A, B, irtc
A = irtc()
DO M = 1,20000
  N = N + M
END DO
B = irtc()
! How many nanoseconds elapsed?
PRINT *, B - A
END
```

---

itime_

The **itime_** subroutine returns the current time in a numerical form containing seconds, minutes, and hours in ITIME_STRUCT.

Format/ Example

```
TYPE IAR
  SEQUENCE
  INTEGER(4) IHR
  INTEGER(4) IMIN
  INTEGER(4) ISEC
END TYPE
TYPE (IAR) ITIME_STRUCT
CALL itime_(ITIME_STRUCT)
```

Procedures

jdate

The **jdate** function returns the current Julian date in yyddd format.

Format/ Example

```
CHARACTER(8) D, jdate
D = jdate()
```

---

lenchr_

The **lenchr_** function stores the length of the character string STR in LENGTH.

Format/ Example

```
INTEGER(4) LENGTH, lenchr_
CHARACTER*(*) STR
LENGTH = lenchr_(STR)
```

---

lnblnk_

The **lnblnk_** function returns the index, INDEX, of the last non-blank character in the string STR. If the string is not found, INDEX is set to 0.

Format/ Example

```
INTEGER(4) INDEX, lnblnk_
CHARACTER(n) STR
INDEX = lnblnk_(STR)
```

Table 16. Service and Utility Procedures (continued)

Procedures

ltime_

The **ltime_** subroutine dissects the system time STIME, which is in seconds, into the array TARRAY containing the GMT where the dissected time is corrected for the local time zone. The data is stored in TARRAY in the following order:
seconds (0 to 59)
minutes (0 to 59)
hours (0 to 23)
day of the month (1 to 31)
month of the year (0 to 11)
year (year = current year - 1900)
day of week (Sunday = 0)
day of year (0 to 365)
daylight saving time (0 or 1)

Format/ Example

```
INTEGER(4) STIME, TARRAY(9)
CALL ltime_(STIME, TARRAY)
```

---

mclock

The **mclock** function returns time accounting information about the current process and its child processes. The returned value is the sum of the current process's user time and the user and system time of all child processes. The unit of measure is one one-hundredth (1/100) of a second.

Format/ Example

```
INTEGER(4) I1, mclock
I1 = mclock()
```

Table 16. Service and Utility Procedures (continued)

Procedures

qsort_

The **qsort_** subroutine performs a parallel quicksort on a one-dimensional array ARRAY whose length LEN is the number of elements in the array with each element having a size of ISIZE, and a user-defined sorting order function COMPAR to sort the elements of the array. Requirements for the COMPAR function are described under the qsort subroutine that is described in the AIX Technical Reference: Base Operating System and Extensions Volume 2.

Format/ Example

```
INTEGER(4) FUNCTION COMPAR_UP(C1, C2)
INTEGER(4) C1, C2
IF (C1.LT.C2) COMPAR_UP = -1
IF (C1.EQ.C2) COMPAR_UP =  0
IF (C1.GT.C2) COMPAR_UP =  1
RETURN
END
SUBROUTINE FOO()
INTEGER(4) COMPAR_UP
EXTERNAL COMPAR_UP
INTEGER(4) ARRAY(8), LEN, ISIZE
DATA ARRAY/0, 3, 1, 2, 9, 5, 7, 4/
LEN = 6
ISIZE = 4
CALL qsort_(ARRAY(3:8), LEN, ISIZE, COMPAR_UP)
! sorting ARRAY(3:8)
PRINT *, ARRAY
! result value is [0, 3, 1, 2, 4, 5, 7, 9]
RETURN
END
```

Procedures

qsort_down

The **qsort_down** subroutine performs a parallel quicksort on a one-dimensional array ARRAY whose length LEN is the number of elements in the array with each element having a size of ISIZE. The result is stored in array ARRAY in descending order. As opposed to **qsort_**, the **qsort_down** subroutine does not require the COMPAR function.

Format/ Example

```
SUBROUTINE FOO()
  INTEGER(4) ARRAY(8), LEN, ISIZE
  DATA ARRAY/0, 3, 1, 2, 9, 5, 7, 4/
  LEN = 8
  ISIZE = 4
  CALL qsort_down(ARRAY, LEN, ISIZE)
  PRINT *, ARRAY
! Result value is [9, 7, 5, 4, 3, 2, 1, 0]
RETURN
END
```

qsort_up

The **qsort_up** subroutine performs a parallel quicksort on a one-dimensional, contiguous array ARRAY whose length LEN is the number of elements in the array with each element having a size of ISIZE. The result is stored in array ARRAY in ascending order. As opposed to **qsort_**, the **qsort_up** subroutine does not require the COMPAR function.

Format/ Example

```
SUBROUTINE FOO()
  INTEGER(4) ARRAY(8), LEN, ISIZE
  DATA ARRAY/0, 3, 1, 2, 9, 5, 7, 4/
  LEN = 8
  ISIZE = 4
  CALL qsort_up(ARRAY, LEN, ISIZE)
  PRINT *, ARRAY
! Result value is [0, 1, 2, 3, 4, 5, 7, 9]
RETURN
END
```

Table 16. Service and Utility Procedures (continued)

Procedures

rtc

The **rtc** function returns a REAL(8) value of the number of seconds since the initial value of the machine's real-time clock.

Format/ Example

```
REAL(8) A, B, rtc
A = rtc()
DO M = 1,20000
  N = N + M
END DO
B = rtc()
! How many seconds elapsed?
PRINT *, B - A
END
```

set_fpscr

The set_fpscr function sets the floating-point status and control register (fpscr) of the processor to the value provided in the FPSCR argument, and returns the value of the register before the change.

Format/ Example

```
USE XLF_FP_UTIL
INTEGER(FPSCR_KIND) FPSCR, OLD_FPSCR

FPSCR=get_fpscr()

! ...  Some changes are made to FPSCR  ...

OLD_FPSCR=set_fpscr(FPSCR)  ! OLD_FPSCR is assigned the value of
                            ! the register before it was
                            ! set with set_fpscr
```

Table 16. Service and Utility Procedures (continued)

Procedures

set_fpscr_flags

The set_fpscr_flags subroutine allows you to set the floating-point status and control register flags you specify in the MASK argument. Flags that you do not specify in MASK remain unaffected. MASK must be of type INTEGER(FPSCR_KIND). You can manipulate the MASK using the intrinsics described in “Integer Bit Model” on page 521.

For more information on the FPSCR constants, see “fpscr constants” on page 641.

Format/ Example

```
USE XLF_FP_UTIL
INTEGER(FPSCR_KIND) SAVED_FPSCR
INTEGER(FP_MODE_KIND) FP_MODE

SAVED_FPSCR = get_fpscr()          ! Saves the current value of
                                   ! the fpscr register.
FP_MODE = fp_trap(FP_TRAP_SYNC)    ! Enables precise trapping.

CALL set_fpscr_flags(TRP_DIV_BY_ZERO) ! Enables trapping of
! ...                                 ! divide-by-zero.
FP_MODE=fp_trap(FP_MODE)              ! Restores initial trap
                                      ! mode.
SAVED_FPSCR=set_fpscr(SAVED_FPSCR)    ! Restores fpscr register.
```

set_round_mode

The set_round_mode function sets the current floating-point rounding mode, and returns the rounding mode before the change. You can set the mode to FP_RND_RN, FP_RND_RZ, FP_RND_RP or FP_RND_RM. For more information on the rounding mode constants, see “fpscr constants” on page 641.

Format/ Example

```
USE XLF_FP_UTIL
INTEGER(FPSCR_KIND) MODE

MODE=set_round_mode(FP_RND_RZ)  ! The rounding mode is set to
                                ! round towards zero.  MODE is
! ...                           ! assigned the previous rounding
                                ! mode.
MODE=set_round_mode(MODE)       ! The rounding mode is restored.
```

Table 16. Service and Utility Procedures (continued)

Procedures

setrteopts

The **setrteopts** subroutine changes the setting of one or more of the run-time options during the execution of a program. See "Setting Runtime Options for Input/Output" in the *User's Guide* for details about the run-time options.

Format/ Example

```
CHARACTER(n) C1
CALL setrteopts (C1)
! For example,
! CALL setrteopts &
!   ('langlvl=90std:cnverr=no')
```

---

sleep_

The **sleep_** subroutine suspends the execution of the current process for SEC seconds.

Format/ Example

```
INTEGER(4) SEC
CALL sleep_(SEC)
```

---

time_

The **time_** function returns the current time (GMT) CURRTIME, in seconds.

Format/ Example

```
INTEGER(4) CURRTIME, time_
CURRTIME = time_()
```

---

timef

The **timef** function returns the elapsed time in milliseconds since the first instance timef was called. The first instance **timef** is called, the value 0.0d0 is returned.

Format/ Example

```
REAL(8) ELAPSED, timef
ELAPSED = timef()
DO M = 1,20000
  A = A ** 2
ENDDO
ELAPSED = TIMEF()
```

Table 16. Service and Utility Procedures (continued)

Procedures

timef_delta

The **timef_delta** function returns the elapsed time in milliseconds since the last instance **timef_delta** was called with its argument set to 0.0 within the same thread. In order to get the correct elapsed time, you must determine which region of a thread you want timed. This region must start with a call to **timef_delta(T0)**, where T0 is initialized (T0=0.0). The next call to **timef_delta** must use the first call's return value as the input argument if the elapsed time is expected.

Format/ Example

```
REAL(8) timef_delta, T0, T1, T2
  T0 = 0.0
    DO I = 1, N         ! Loop 1
      H = I + 1000
    END DO
  T1 = timef_delta(T0)
    DO I = 1, N         ! T1 gives the
      M = I + 2000      ! starting time
    END DO              ! of loop 2
  T2 = timef_delta(T1)
    DO I = 1, N         ! T2 gives the
      M = I + 3000      ! elapsed time
    END DO              ! of loop 2
```

umask_

The **umask_** function sets the file mode creation mask to CMASK. The returned value, LASTMASK, is the previous value of the file mode creation mask.

Format/ Example

```
INTEGER(4) CMASK, LASTMASK, umask_
LASTMASK = umask_ (CMASK)
```

usleep_

The **usleep_** function suspends the execution of the current process for an interval of MSEC microseconds. The returned value, IS_SLEEP, is 0 if the function is successful, or an error number otherwise.

Format/ Example

```
INTEGER(4) IS_SLEEP, MSEC, usleep_
IS_SLEEP = usleep_ (MSEC)
```

Table 16. Service and Utility Procedures (continued)

Procedures

xl_ _trbk

The **xl_ _trbk** subroutine provides a traceback starting from the invocation point. **xl_ _trbk** can be called from your code, although not from signal handlers. The subroutine requires no parameters.

Format/ Example

```
INTEGER res, n
IF (n .EQ. 1) THEN
   res=1
   CALL XL_ _TRBK()
ELSE
   res=n * FACTORIAL(n-1)
ENDIF
```

Chapter 14. OpenMP Execution Environment Routines and Lock Routines

The OpenMP specification provides a number of routines which allow you to control and query the parallel execution environment.

Parallel threads created by the run-time environment through the OpenMP interface are considered independent of the threads you create and control using calls to the **Fortran Pthreads** library module. That is to say, references within the following descriptions to ″serial portions of the program″ should be understood to mean portions of the program that are executed by only one of the threads that have been created by the run-time environment. For example, you can create multiple threads by using **f_pthread_create**. However, if you then call **omp_get_num_threads** from outside of an OpenMP parallel block, or from within a serialized nested parallel region, the function will return **1**, regardless of the number of threads that are currently executing.

The OpenMP execution environment routines are:

- **omp_get_dynamic**: see "omp_get_dynamic" on page 668
- **omp_get_max_threads**: see "omp_get_max_threads" on page 669
- **omp_get_nested**: see "omp_get_nested" on page 669
- **omp_get_num_procs**: see "omp_get_num_procs" on page 669
- **omp_get_num_threads**: see "omp_get_num_threads" on page 670
- **omp_get_thread_num**: see "omp_get_thread_num" on page 671
- **omp_in_parallel**: see "omp_in_parallel" on page 672
- **omp_set_dynamic**: see "omp_set_dynamic" on page 673
- **omp_set_nested**: see "omp_set_nested" on page 674
- **omp_set_num_threads**: see "omp_set_num_threads" on page 675

The OpenMP run-time library also includes a set of general-purpose locking routines, which are listed below. You must only lock variables through these routines. For all routines, the lock variable should be an integer of a **KIND** large enough to hold an address.

For example, in 32–bit compilation bit mode, the lock variable should be declared as **INTEGER (4)** or larger. If you are compiling your program in 64–bit mode, you should declare the lock variable as **INTEGER (8)**.

OpenMP provides the following lock routines:

- **omp_destroy_lock**: see "omp_destroy_lock" on page 668

- **omp_init_lock**: see “omp_init_lock” on page 672
- **omp_set_lock**: see “omp_set_lock” on page 674
- **omp_test_lock**: see “omp_test_lock” on page 675
- **omp_unset_lock**: see “omp_unset_lock” on page 676

Note: You can define and implement your own versions of the OpenMP routines. However, by default, the compiler will substitute the XL Fortran versions of the OpenMP routines regardless of the existence of other implementations, unless you specify the **–qnoswapomp** compiler option. For more information, see *User's Guide*.

Table 17. OpenMP Execution Environment Routines and Lock Routines

Procedures

---

omp_destroy_lock

This subroutine dissassociates a given lock variable from all locks. You have to use **omp_init_lock** to reinitialize a lock variable that has been destroyed with a call to **omp_destroy_lock** before using it again as a lock variable.

Note: If you call **omp_destroy_lock** with a lock variable that has not been initialized, the result of the call is undefined.

Format/ Example

```
INTEGER(8) LOCK
CALL omp_destroy_lock(LOCK)
```

For an example of how to use omp_destroy_lock, see “omp_init_lock” on page 672

---

omp_get_dynamic

The **omp_get_dynamic** function returns **.TRUE.** if dynamic thread adjustment by the run-time environment is enabled, and **.FALSE.** otherwise.

Format/ Example

```
LOGICAL omp_get_dynamic, LVAR
LVAR = omp_get_dynamic()
```

Table 17. OpenMP Execution Environment Routines and Lock Routines (continued)

Procedures

omp_get_max_threads
 	
This function returns the maximum number of threads that can execute concurrently in a single parallel region. The return value is equal to the maximum value that can be returned by the **omp_get_num_threads** function. If you use **omp_set_num_threads** to change the number of threads, subsequent calls to **omp_get_max_threads** will return the new value.
 	
The function has global scope, which means that the maximum value it returns applies to all functions, subroutines, and compilation units in the program. It returns the same value whether executing from a serial or parallel region.
 	
You can use **omp_get_max_threads** to allocate maximum-sized data structures for each thread when you have enabled dynamic thread adjustment by passing **omp_set_dynamic** an argument which evaluates to **.TRUE.**
 	
Format/ Example
 	
`INTEGER MAX_THREADS, omp_get_max_threads`
`MAX_THREADS = omp_get_max_threads()`

omp_get_nested
 	
The **omp_get_nested** function returns **.TRUE.** if nested parallelism is enabled and **.FALSE.** if nested parallelism is disabled.
 	
Format/ Example
 	
`LOGICAL omp_get_nested, LVAR`
`LVAR = omp_get_nested()`

omp_get_num_procs
 	
The **omp_get_num_procs** function returns the number of online processors on the machine.
 	
Format/ Example
 	
`INTEGER NUM_PROCS, omp_get_num_procs`
`NUM_PROCS = omp_get_num_procs()`

Procedures

omp_get_num_threads

The **omp_get_num_threads** function returns the number of threads in the team currently executing the parallel region from which it is called. The function binds to the closest enclosing **PARALLEL** directive.

The **omp_set_num_threads** subroutine and the **OMP_NUM_THREADS** environment variable control the number of threads in a team. If you do not explicitly set the number of threads, the run-time environment will use the number of online processors on the machine by default.

If you call **omp_get_num_threads** from a serial portion of your program or from a nested parallel region that is serialized, the function returns 1.

Format/ Example

```
      INTEGER N1, N2, omp_get_num_threads

      N1 = omp_get_num_threads()
      PRINT *, N1
!$OMP PARALLEL PRIVATE(N2)
      N2 = omp_get_num_threads()
      PRINT *, N2
!$OMP END PARALLEL
```

The **omp_get_num_threads** call returns 1 in the serial section of the code, so N1 is assigned the value 1. N2 is assigned the number of threads in the team executing the parallel region, so the output of the second print statement will be an arbitrary number less than or equal to the value returned by **omp_get_max_threads**.

Procedures

omp_get_thread_num

This function returns the number of the currently executing thread within the team. The number returned will always be between 0 and NUM_PARTHDS - 1. The master thread of the team returns a value of 0.

If you call **omp_get_thread_num** from within a serial region, from within a serialized nested parallel region, or from outside the dynamic extent of any parallel region, this function will return a value of 0.

This function binds to the closest **PARALLEL**, **PARALLEL DO**, or **PARALLEL SECTIONS** directive that encloses it.

Format/ Example

```
      INTEGER NP, omp_get_thread_num

!$OMP PARALLEL PRIVATE(NP)
      NP = omp_get_thread_num()
      CALL WORK(NP)
!$OMP MASTER
      NP = omp_get_thread_num()
      CALL WORK(NP)
!$OMP END MASTER
!$OMP END PARALLEL
      END

      SUBROUTINE WORK(THD_NUM)
      INTEGER THD_NUM
      PRINT *, THD_NUM
      END
```

Table 17. OpenMP Execution Environment Routines and Lock Routines (continued)

Procedures

omp_in_parallel

The **omp_in_parallel** function returns **.TRUE.** if you call it from the dynamic extent of a region executing in parallel and returns **.FALSE.** otherwise. If you call **omp_in_parallel** from a region that is serialized but nested within the dynamic extent of a region executing in parallel, the function will still return **.TRUE.**. (Nested parallel regions are serialized by default. See "omp_set_nested" on page 674 and the environment variable **OMP_NESTED** in *XL Fortran for AIX User's Guide*, Chapter 4, for more information.)

Format/ Example

```
      INTEGER N, M, omp_in_parallel
      N = 4
      M = 3
      PRINT*, omp_in_parallel()
!$OMP PARALLEL DO
      DO I = 1,N
!$OMP   PARALLEL DO
        DO J=1, M
          PRINT *, omp_in_parallel()
        END DO
!$OMP   END PARALLEL DO
      END DO
!$OMP END PARALLEL DO
```

The first call to **omp_in_parallel** returns **.FALSE.** because the call is outside the dynamic extent of any parallel region. The second call returns **.TRUE.**, even if the nested **PARALLEL DO** loop is serialized, because the call is still inside the dynamic extent of the outer **PARALLEL DO** loop.

omp_init_lock

The **omp_init_lock** subroutine initializes a lock and associates it with the lock variable passed in as a parameter. After the call to **omp_init_lock**, the initial state of the lock variable is unlocked.

Note: If you call this routine with a lock variable that you have already initialized, the result of the call is undefined.

Format/ Example

```
      INTEGER LCK          !THIS VARIABLE SHOULD BE POINTER SIZED
      INTEGER ID, omp_get_thread_num
      CALL omp_init_lock(LCK)
!$OMP PARALLEL SHARED(LCK), PRIVATE(ID)
      ID = omp_get_thread_num()
      CALL omp_set_lock(LCK)
      PRINT *,'MY THREAD ID IS', ID
      CALL omp_unset_lock(LCK)
!$OMP END PARALLEL
      CALL omp_destroy_lock(LCK)
```

In the above example, one at a time, the threads gain ownership of the lock associated with the lock variable LCK, print the thread ID, and release ownership of the lock.

Table 17. OpenMP Execution Environment Routines and Lock Routines (continued)

Procedures

omp_set_dynamic
 	
The **omp_set_dynamic** subroutine enables or disables dynamic adjustment, by the run-time environment, of the number of threads available to execute parallel regions.
 	
If you call **omp_set_dynamic** with a *scalar_logical_expression* that evaluates to **.TRUE.**, the run-time environment can automatically adjust the number of threads that are used to execute subsequent parallel regions to obtain the best use of system resources. The number of threads you specify using **omp_set_num_threads** becomes the maximum, not exact, thread count.
 	
If you call the subroutine with a *scalar_logical_expression* which evaluates to **.FALSE.**, dynamic adjustment of the number of threads is disabled. The run-time environment cannot automatically adjust the number of threads used to execute subsequent parallel regions. The value you pass to **omp_set_num_threads** becomes the exact thread count.
 	
By default, dynamic thread adjustment is enabled. If your code depends on a specific number of threads for correct execution, you should explicitly disable dynamic threads.
 	
Note: The number of threads remains fixed for each parallel region. The **omp_get_num_threads** function returns that number.
 	
This subroutine has precedence over the **OMP_DYNAMIC** environment variable.
 	
Format/ Example
`LOGICAL LVAR`
`CALL omp_set_dynamic(LVAR)`

Procedures

omp_set_lock

The **omp_set_lock** subroutine forces the calling thread to wait until the specified lock is available before executing subsequent instructions. The calling thread is given ownership of the lock when it becomes available.

Note: If you call this routine with a lock variable that has not been initialized, the result of the call is undefined. Also, if a thread that owns a lock tries to lock it again by issuing a call to **omp_set_lock**, it will produce a deadlock.

Format/ Example

```
      INTEGER LCK_X      !THIS VARIABLE SHOULD BE POINTER SIZED
      CALL omp_init_lock (LCK_X)
!$OMP PARALLEL PRIVATE (I), SHARED (A, X)
!$OMP DO
      DO I = 3, 100
        A(I) = I * 10
        CALL omp_set_lock (LCK_X)
        X = X + A(I)
        CALL omp_unset_lock (LCK_X)
      END DO
!$OMP END DO
!$OMP END PARALLEL
      CALL omp_destroy_lock (LCK)
```

In this example, the lock variable LCK_X is used to avoid race conditions when updating the shared variable X. By setting the lock before each update to X and unsetting it after the update, you ensure that only one thread is updating X at a given time.

omp_set_nested

The **omp_set_nested** subroutine enables or disables nested parallelism.

If you call the subroutine with a *scalar_logical_expression* that evaluates to **.FALSE.**, nested parallelism is disabled. Nested parallel regions are serialized, and they are executed by the current thread. This is the default setting.

If you call the subroutine with a *scalar_logical_expression* that evaluates to **.TRUE.**, nested parallelism is enabled. Parallel regions that are nested can deploy additional threads to the team. It is up to the run-time environment to determine whether additional threads should be deployed. Therefore, the number of threads used to execute parallel regions may vary from one nested region to the next.

This subroutine takes precedence over the **OMP_NESTED** environment variable.

Format/ Example

```
      LOGICAL LVAR
      CALL omp_set_nested(LVAR)
```

Table 17. OpenMP Execution Environment Routines and Lock Routines (continued)

Procedures

omp_set_num_threads

The **omp_set_num_threads** subroutine tells the run-time environment how many threads to use in the next parallel region. The *scalar_integer_expression* that you pass to the subroutine is evaluated, and its value is used as the number of threads. If you have enabled dynamic adjustment of the number of threads (see “omp_set_dynamic” on page 673), **omp_set_num_threads** sets the maximum number of threads to use for the next parallel region. The run-time environment then determines the exact number of threads to use. However, when dynamic adjustment of the number of threads is disabled, **omp_set_num_threads** sets the exact number of threads to use in the next parallel region.

This subroutine takes precedence over the **OMP_NUM_THREADS** environment variable.

Note: If you call this subroutine from the dynamic extent of a region executing in parallel, the behavior of the subroutine is undefined.

Format/ Example

```
      INTEGER(8) NUM_THREADS
      CALL omp_set_num_threads(NUM_THREADS)
```

---

omp_test_lock

The **omp_test_lock** function attempts to set the lock associated with the specified lock variable. It returns **.TRUE.** if it was able to set the lock and **.FALSE.** otherwise. In either case, the calling thread will continue to execute subsequent instructions in the program.

Note: If you call **omp_test_lock** with a lock variable that has not yet been initialized, the result of the call is undefined.

Format/ Example

```
      INTEGER LCK        !THIS VARIABLE SHOULD BE POINTER SIZED
      INTEGER ID, omp_get_thread_num
      LOGICAL omp_test_lock
      CALL omp_init_lock (LCK)
!$OMP PARALLEL SHARED(LCK), PRIVATE(ID)
      ID = omp_get_thread_num()
      DO WHILE (.NOT. omp_test_lock(LCK))
        CALL WORK_A (ID)
      END DO
      CALL WORK_B (ID)
!$OMP END PARALLEL
      CALL omp_destroy_lock (LCK)
```

In this example, a thread repeatedly executes WORK_A until it can set LCK, the lock variable. When it succeeds in setting the lock variable, it executes WORK_B.

Table 17. OpenMP Execution Environment Routines and Lock Routines (continued)

Procedures

omp_unset_lock
 	
This subroutine causes the executing thread to release ownership of the specified lock. The lock can then be set by another thread as required.
 	
Note: The behavior of the **omp_unset_lock** subroutine is undefined if either:
• The calling thread does not own the lock specified, or
• The routine is called with a lock variable that has not been initialized.
 	
Format/ Example
`INTEGER(8) LOCK`
`CALL omp_unset_lock(LOCK)`
 	
For an example of how to use **omp_unset_lock**, see “omp_set_lock” on page 674.

Chapter 15. Pthreads Library Module

The Pthreads Library Module (**f_pthread**) is a Fortran 90 module that defines data types and routines to make it easier to interface with the AIX pthreads library. The AIX pthreads library is used to parallelize and thread-safe your code. The **f_pthread** library module naming convention is the use of the prefix **f_** before the corresponding AIX pthreads library routine name or type definition name.

The pthreads library for AIX 4.1 and AIX 4.2 complies with the POSIX standard Draft 7, while on AIX 4.3, the pthreads library complies with the POSIX standard Draft 10. There are discrepancies in these two pthreads libraries. Since the pthread library module provides Fortran interfaces to the AIX pthreads library, a separate Fortran pthreads library module is desirable to match the AIX 4.3 pthreads library interfaces.

AIX Version 4.3 supports two versions of the pthreads library module interface: Draft 10 (the default) and Draft 7. Depending on which invocation command you use, you can compile and link your programs with either the Draft 10 interface libraries or the Draft 7 interface libraries. For more information about how to do this, see ″Compiling Multi-Threaded Programs″ and ″Linking Multi-Threaded Programs Using the ″ld″ Command″ in the User's Guide.

In general, there is a one-to-one corresponding relationship between the procedures in the Fortran 90 module **f_pthread** and the library routines contained in the AIX pthreads library. However, some of the pthread routines have no corresponding procedures in this module because they are not supported on AIX. One example of these routines is the thread stack address option. There are also some non-pthread interfacing routines contained in the **f_pthread** library module. The **f_maketime** routine is one example and is included to return an absolute time in a **f_timespec** derived type variable.

Most of the routines return an integer value. A return value of **0** will always indicate that the routine call did not result in any error. Any non-zero return value indicates an error. Each error code has a corresponding definition of a system error code in Fortran. These error codes are available as Fortran integer constants. The naming of these error codes in Fortran is consistent with the corresponding AIX error code names. For example, **EINVAL** is the Fortran constant name of the error code **EINVAL** on AIX. For a complete list of these error codes, refer to the file **/usr/include/sys/errno.h** on AIX.

© Copyright IBM Corp. 1990, 2000

For more information about the system calls corresponding to the Fortran Pthreads library calls, see the AIX Operating System documentation.

The Pthreads Data Structures

f_pthread_attr_t
f_pthread_cond_t
f_pthread_condattr_t
f_pthread_key_t
f_pthread_mutex_t
f_pthread_mutexattr_t
f_pthread_once_t
f_pthread_t
f_sched_param
f_timespec

Functions That Perform Operations on Thread Attribute Objects

f_pthread_attr_destroy
f_pthread_attr_getdetachstate
f_pthread_attr_getinheritsched
f_pthread_attr_getschedparam
f_pthread_attr_getschedpolicy
f_pthread_attr_getscope
f_pthread_attr_getstacksize
f_pthread_attr_init
f_pthread_attr_setdetachstate
f_pthread_attr_setinheritsched
f_pthread_attr_setschedparam
f_pthread_attr_setschedpolicy
f_pthread_attr_setscope
f_pthread_attr_setstacksize

Functions and Subroutines That Perform Operations on Thread

f_pthread_cancel
f_pthread_cleanup_pop
f_pthread_cleanup_push
f_pthread_create
f_pthread_equal
f_pthread_exit
f_pthread_getschedparam
f_pthread_join
f_pthread_kill
f_pthread_self
f_pthread_setschedparam

Functions That Perform Operations on Mutex Attribute Objects

f_pthread_mutexattr_destroy
f_pthread_mutexattr_getprioceiling
f_pthread_mutexattr_getprotocol
f_pthread_mutexattr_getpshared
f_pthread_mutexattr_init
f_pthread_mutexattr_setprioceiling
f_pthread_mutexattr_setprotocol
f_pthread_mutexattr_setpshared

Functions That Perform Operations on Mutex Objects

f_pthread_mutex_destroy
f_pthread_mutex_getprioceiling
f_pthread_mutex_init
f_pthread_mutex_lock
f_pthread_mutex_setprioceiling
f_pthread_mutex_trylock
f_pthread_mutex_unlock

Functions That Perform Operations on Attribute Objects of Condition Variables

f_pthread_condattr_destroy
f_pthread_condattr_getpshared
f_pthread_condattr_init
f_pthread_condattr_setpshared

Functions That Perform Operations on Condition Variable Objects

f_maketime
f_pthread_cond_broadcast
f_pthread_cond_destroy
f_pthread_cond_init
f_pthread_cond_signal
f_pthread_cond_timedwait
f_pthread_cond_wait

Functions That Perform Operations on Thread-Specific Data

f_pthread_getspecific
f_pthread_key_create
f_pthread_key_delete
f_pthread_setspecific

Functions and Subroutines That Perform Operations to Control Thread Cancelability

f_pthread_setcancelstate
f_pthread_setcanceltype
f_pthread_testcancel

Functions That Perform Operations for One-Time Initialization

f_pthread_once

Table 18. Fortran Pthreads Library Module

Functions, Subroutines, and Data Structures

---

f_maketime

This function accepts an integer value specifying a delay in seconds and returns an **f_timespec** type object containing the absolute time, which is **delay** seconds from the calling moment.

Return Value:

The absolute time, which is **delay** seconds from the calling moment, is returned.

Example:

```
type(f_timespec) function f_maketime(delay)
    integer(4), intent(in):: delay
end function
```

---

f_pthread_attr_destroy

This function must be called to destroy any previously initialized thread attribute objects when they will no longer be used. Threads that were created with this attribute object will not be affected in any way by this action. Memory that was allocated when it was initialized will be recollected by the system.

Return Codes:

If errors are detected during the execution of this function, the following error code will be returned:

EINVAL The argument **attr** is invalid.

Example:

```
integer function f_pthread_attr_destroy(attr)
    type(f_pthread_attr_t), intent(inout):: attr
end function
```

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_attr_getdetachstate

This function can be used to query the setting of the detach state attribute in the thread attribute object **attr**. The current setting will be returned through argument **detach**.

Argument **detach** will contain one of the following values:

PTHREAD_CREATE_DETACHED:
when a thread attribute object of this attribute setting is used to create a new thread, the newly created thread will be in detached state. This is the system default.

PTHREAD_CREATE_UNDETACHED:
when a thread attribute object of this attribute setting is used to create a new thread, the newly created thread will be in undetached state.

For more information about these thread states, refer to the AIX Operating System documentation.

Return Codes:

If errors are detected during the execution of this function, the following error code will be returned:

EINVAL The argument **attr** is invalid.

Example:

```
integer function f_pthread_attr_getdetachstate(attr, detach)
    type(f_pthread_attr_t), intent(in):: attr
    integer(4), intent(out):: detach
end function
```

f_pthread_attr_getguardsize

This function is used to get the *guardsize* attribute in the thread attribute object *attr*. The current setting of the attribute will be returned through the argument *guardsize*.

Return Codes:

If this function completes successfully, value 0 is returned. Otherwise, the following error will be returned:

```
EINVAL  The argument attr is invalid.
```

Example:

```
integer(4) function f_pthread_attr_getguardsize(attr, guardsize)
type(f_pthread_attr_t), intent(in):: attr
integer(kind=REGISTER_SIZE), intent(out):: guardsize
end function f_pthread_attr_getguardsize
```

Functions, Subroutines, and Data Structures

f_pthread_attr_getinheritsched

This function can be used to query the inheritance scheduling attribute in the thread attribute object **attr**. The current setting will be returned through the argument **inherit**.

Argument **inherit** will contain one of the following values:

PTHREAD_INHERIT_SCHED: indicating that newly created threads will inherit the scheduling property of the parent thread and ignore the scheduling property of the thread attribute object used to create them.

PTHREAD_EXPLICIT_SCHED: the scheduling property in the thread attribute object will be assigned to the newly created threads when it is used to create them.

Return Codes:

If errors are detected during the execution of this function, one of the following error codes will be returned:

EINVAL The argument **attr** is invalid.

ENOSYS The POSIX priority scheduling option is not implemented on AIX.

Example:

```
integer function f_pthread_attr_getinheritsched(attr, inherit)
    type(f_pthread_attr_t), intent(in):: attr
    integer(4), intent(out):: inherit
end function
```

f_pthread_attr_getschedparam

This function can be used to query the scheduling property setting in the thread attribute object **attr**. The current setting will be returned in the argument **param**. See the AIX system documentation for more information on the scheduling property setting.

Return Codes:

If errors are detected during the execution of this function, one of the following error codes will be returned:

EINVAL The argument **attr** is invalid.

ENOSYS The POSIX priority scheduling option is not implemented on AIX.

Example:

```
integer function f_pthread_attr_getschedparam(attr, param)
    type(f_pthread_attr_t), intent(in):: attr
    type(f_sched_param), intent(out):: param
end function
```

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_attr_getschedpolicy

This function can be used to query the scheduling policy attribute setting in the attribute object **attr**. The current setting of the scheduling policy will be returned in the argument **policy**. The valid scheduling policies on AIX can be found in the AIX Operating System documentation.

Return Codes:

If errors are detected during the execution of this function, one of the following error codes will be returned:

EINVAL
The argument **attr** is invalid.

ENOSYS
The POSIX priority scheduling option is not implemented on AIX.

Example:

```
integer function f_pthread_attr_getschedpolicy(attr, policy)
    type(f_pthread_attr_t), intent(in):: attr
    integer(4), intent(out):: policy
```

f_pthread_attr_getscope

This function can be used to query the current setting of the scheduling scope attribute in the thread attribute object **attr**. The current setting will be returned through the argument **scope**.

Argument **scope** will contain one of the following values:

PTHREAD_SCOPE_SYSTEM: the thread will compete for system resources on a system wide scope.

PTHREAD_SCOPE_PROCESS: the thread will compete for system resources locally within the owning process.

Return Codes:

If errors are detected during the execution of this function, one of the following error codes will be returned:

EINVAL The argument **attr** is invalid.

ENOSYS The POSIX priority scheduling option is not implemented on AIX.

Example:

```
integer function f_pthread_attr_getscope(attr, scope)
    type(f_pthread_attr_t), intent(in):: attr
    integer(4), intent(out):: scope
end function
```

Functions, Subroutines, and Data Structures

f_pthread_attr_getstackaddr

This function is used to get the *stackaddr*attribute in the thread attribute object *attr*. The current setting of the attribute will be returned through the argument *stackaddr*. The type of the argument *stackaddr* is **integer pointer**. The *stackaddr* attribute specifies the stack address of a thread created with this attributes object.

Return Codes:

If this function completes successfully, value 0 is returned. Otherwise, the following error will be returned:

```
EINVAL The argument attr is invalid.
```

Example:

```
integer(4) function f_pthread_attr_getstackaddr(attr, stackaddr)
type(f_pthread_attr_t), intent(in):: attr
integer(4) int_template
pointer (stackaddr, int_template)
intent(out) stackaddr
end function f_pthread_attr_getstackaddr
```

f_pthread_attr_getstacksize

This function can be used to query the current stack size attribute setting in the attribute object **attr**. If this function executes successfully, the stack size in bytes will be returned in argument **ssize**.

Return Codes:

If errors are detected during the execution of this function, one of the following error codes will be returned:

EINVAL
: The argument **attr** is invalid.

ENOSYS
: The POSIX stack size option is not implemented on AIX.

Example:

```
integer function f_pthread_attr_getstacksize(attr, ssize)
    type(f_pthread_attr_t), intent(in):: attr
    integer(4), intent(out):: ssize
end function
```

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_attr_init

This function must be called to create and initialize the pthread attribute object **attr** before it can be used in any way. It will be filled with system default thread attribute values. After it is initialized, certain pthread attributes can be changed and/or set through attribute access procedures. Once initialized, this attribute object can be used to create a thread with the intended attributes. Refer to the AIX Operating System documentation for more information on the default attributes.

Return Codes:

If errors are detected during the execution of this function, one of the following error codes will be returned:

EINVAL The argument **attr** is invalid.

ENOMEM There is insufficient memory to create this attribute object.

Example:

```
integer function f_pthread_attr_init(attr)
    type(f_pthread_attr_t), intent(out):: attr
end function
```

f_pthread_attr_setdetachstate

This function can be used to set the detach state attribute in the thread attribute object **attr**.

Argument **detach** must contain one of the following values:

PTHREAD_CREATE_DETACHED:
when a thread attribute object of this attribute setting is used to create a new thread, the newly created thread will be in detached state. This is the system default setting.

PTHREAD_CREATE_UNDETACHED:
when a thread attribute object of this attribute setting is used to create a new thread, the newly created thread will be in undetached state.

For more information about these thread states, refer to the AIX Operating System documentation.

Return Codes:

If errors are detected during the execution of this function, the following error code will be returned:

EINVAL The argument **attr** or **detach** is invalid.

Example:

```
integer function f_pthread_attr_setdetachstate(attr, detach)
    type(f_pthread_attr_t), intent(inout):: attr
    integer(4), intent(in):: detach
end function
```

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_attr_setguardsize

This function is used to set the *guardsize*attribute in the thread attributes object *attr*. The new value of this attribute is obtained from the argument *guardsize*. If *guardsize* is zero, a guard area will not be provided for threads created with *attr*. If *guardsize* is greater than zero, a guard area of at least size*guardsize*bytes is provided for each thread created with*attr*. For more information about *guardsize*, refer to the AIX Operating System documentation.

Return Codes:

If this function completes successfully, value 0 is returned. Otherwise, one of the following errors will be returned:

```
EINVAL  The argument attr is invalid.
EINVAL  The argument guardsize is invalid.
```

Example:

```
integer(4) function f_pthread_attr_setguardsize(attr, guardsize)
type(f_pthread_attr_t), intent(inout):: attr
integer(kind=REGISTER_SIZE), intent(in):: guardsize
end function f_pthread_attr_setguardsize
```

Functions, Subroutines, and Data Structures

f_pthread_attr_setinheritsched

This function can be used to set the inheritance attribute of the thread scheduling property in the thread attribute object **attr**.

Argument **inherit** must contain one of the following values:

PTHREAD_INHERIT_SCHED: indicating that newly created threads will inherit the scheduling property of the parent thread and ignore the scheduling property of the thread attribute object used to create them.

PTHREAD_EXPLICIT_SCHED: the scheduling property in the thread attribute object will be assigned to the newly created threads when it is used to create them.

Return Codes:

If errors are detected during the execution of this function, one of the following error codes will be returned:

EINVAL
The argument **attr** is invalid.

ENOSYS
The POSIX priority scheduling option is not implemented on AIX.

ENOTSUP
The value of argument **inherit** is not supported.

Example:

```
integer function f_pthread_attr_setinheritsched(attr, inherit)
    type(f_pthread_attr_t), intent(inout):: attr
    integer(4), intent(in):: inherit
end function
```

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures
f_pthread_attr_setschedparam

This function can be used to set the scheduling property attribute in the thread attribute object **attr**. Threads created with this new attribute object will assume the scheduling property of argument **param** if they are not inherited from the creating thread. The sched_priority field in argument **param** indicates the thread's scheduling priority. The priority field must assume a value in the range of 1-127, where *127* is the most favored scheduling priority while *1* is the least.

Return Codes:

If errors are detected during the execution of this function, one of the following error codes will be returned:

EINVAL
The argument **attr** is invalid.

ENOSYS
The POSIX priority scheduling option is not implemented on AIX.

ENOTSUP
The value of argument **param** is not supported.

Example:

```
integer function f_pthread_attr_setschedparam(attr, param)
    type(f_pthread_attr_t), intent(inout):: attr
    type(f_sched_param), intent(in):: param
end function
```

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_attr_setschedpolicy

After the attribute object is set by this function, threads created with this attribute object will assume the set scheduling policy if the scheduling property is not inherited from the creating thread.

Argument **policy** must contain one of the following constants:

SCHED_FIFO: indicating a first-in first-out thread scheduling policy.

SCHED_RR: indicating a round-robin scheduling policy.

SCHED_OTHER: the default scheduling policy.

Return Codes:

If errors are detected during the execution of this function, one of the following error codes will be returned:

EINVAL
The argument **attr** is invalid.

ENOSYS
The POSIX priority scheduling option is not implemented on AIX.

ENOTSUP
The value of argument **policy** is not supported.

Example:

```
integer function f_pthread_attr_setschedpolicy(attr, policy)
    type(f_pthread_attr_t), intent(inout):: attr
    integer(4), intent(in):: policy
end function
```

Functions, Subroutines, and Data Structures

f_pthread_attr_setscope

This function can be used to set the contention scope attribute in the thread attribute object **attr**.

Argument **scope** must contain one of the following values:

PTHREAD_SCOPE_SYSTEM: the thread will compete for system resources on a system wide scope.

PTHREAD_SCOPE_PROCESS: the thread will compete for system resources locally within the owning process.

Return Codes:

If errors are detected during the execution of this function, one of the following error codes will be returned:

EINVAL The argument **attr** is invalid.

ENOSYS The POSIX priority scheduling option is not implemented on AIX.

ENOTSUP The value of argument **scope** is not supported.

Example:

```
integer function f_pthread_attr_setscope(attr, scope)
    type(f_pthread_attr_t), intent(inout):: attr
    integer(4), intent(in):: scope
end function
```

f_pthread_attr_setstackaddr

This function is used to set the *stackaddr* attribute in the thread attributes object *attr*. The new value of this attribute is obtained from the argument *stackaddr*. The type of the argument *stackaddr*is integer pointer. The *stackaddr* attribute specifies the stack address of a thread created with this attributes object.

Return Codes:

If this function completes successfully, value 0 is returned. Otherwise, the following error will be returned:

```
EINVAL The argument attr is invalid.
```

Example:

```
integer(4) function f_pthread_attr_setstackaddr(attr, stackaddr)
type(f_pthread_attr_t), intent(inout):: attr
integer(4) int_template
pointer (stackaddr, int_template)
intent(in) stackaddr
end function f_pthread_attr_setstackaddr
```

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_attr_setstacksize

This function can be used to set the stack size attribute in the pthread attribute object **attr**. Argument **ssize** is an integer indicating the stack size desired in bytes. When a thread is created using this attribute object, the system will allocate a minimum stack size of **ssize** bytes.

Return Codes:

If errors are detected during the execution of this function, one of the following error codes will be returned:

EINVAL The argument **attr** or **ssize** is invalid.

ENOSYS The POSIX stack size option is not implemented on AIX.

Example:

```
integer function f_pthread_attr_setstacksize(attr, ssize)
    type(f_pthread_attr_t), intent(inout):: attr
    integer(4), intent(in):: ssize
end function
```

f_pthread_attr_t

A derived data type whose components are all private. Any object of this type should be manipulated only through the appropriate interfaces provided in this module.

This data type corresponds to the POSIX **pthread_attr_t**, which is the type of thread attribute object.

f_pthread_cancel

This function can be used to cancel a target thread. How this cancelation request will be processed depends on the state of the cancelability of the target thread. The target thread is identified by argument **thread**. If the target thread is in deferred-cancel state, this cancelation request will be put on hold until the target thread reaches its next cancelation point. If the target thread disables its cancelability, this request will be put on hold until it is enabled again. If the target thread is in async-cancel state, this request will be acted upon immediately. For more information about thread cancelation and concerns about security, refer to the AIX Operating System documentation.

Return Codes:

If errors are detected during the execution of this function, the following error code will be returned:

EINVAL
The argument **thread** is invalid.

Example:

```
integer function f_pthread_cancel(thread)
    type(f_pthread_t), intent(inout):: thread
end function
```

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures
f_pthread_cleanup_pop

This subroutine should be paired with **f_pthread_cleanup_push** in using the cleanup stack for thread safety. If the supplied argument **exec** contains a non-zero value, the last pushed cleanup function will be popped from the cleanup stack and executed, with the argument **arg** (from the last **f_pthread_cleanup_push**) passed to the cleanup function.

If **exec** contains a zero value, the last pushed cleanup function will be popped from the cleanup stack, but will not be executed.

Return Codes:

There is no return value from this subroutine.

Example:

```
subroutine f_pthread_cleanup_pop(exec)
    integer(4), intent(in):: exec
end subroutine
```

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_cleanup_push

This function can be used to register a cleanup subroutine for the calling thread. In case of an unexpected termination of the calling thread, the system will automatically execute the cleanup subroutine in order for the calling thread to terminate safely. The argument **cleanup** must be a subroutine expecting exactly one argument. If it is executed, the argument **arg** will be passed to it as the actual argument.

Note that argument **arg** is a generic argument that can be of any type and any rank with the limitations detailed on page 727.

For a normal execution path, this function must be paired with a call to **f_pthread_cleanup_pop**.

The argument **flag** must be used to convey the property of argument **arg** exactly to the system.

Argument **flag** can assume a value that is one of, or a combination of, the following constants:

FLAG_CHARACTER:
: if the entry subroutine **cleanup** expects an argument of type **CHARACTER** in any way or any form, this flag value must be included to indicate this fact. However, if the subroutine expects a Fortran 90 pointer pointing to an argument of type **CHARACTER**, the **FLAG_DEFAULT** value should be included instead.

FLAG_ASSUMED_SHAPE:
: if the entry subroutine **cleanup** has a dummy argument that is an assumed-shape array of any rank, this flag value must be included to indicate this fact.

FLAG_DEFAULT:
: otherwise, this flag value is needed.

Return Codes:

If errors are detected during the execution of this function, one of the following error codes will be returned:

ENOMEM The system cannot allocate memory to push this routine.

EAGAIN The system cannot allocate resources to push this routine.

EINVAL The argument **flag** is invalid.

Example:

```
integer function f_pthread_cleanup_push(cleanup, flag, arg)
    external cleanup
    integer(4), intent(in):: flag
end function
```

Functions, Subroutines, and Data Structures

f_pthread_cond_broadcast

This function will unblock all threads waiting on the condition variable **cond**. If there is no thread waiting on this condition variable, the function will still succeed, but the next caller to **f_pthread_cond_wait** will be blocked, and will wait on the condition variable **cond**.

Return Codes:

If errors are detected during the execution of this function, the following error code will be returned:

EINVAL The argument **cond** is invalid.

Example:

```
integer function f_pthread_cond_broadcast(cond)
    type(f_pthread_cond_t), intent(inout):: cond
end function
```

f_pthread_cond_destroy

This function can be used to destroy those condition variables that are no longer required. The target condition variable is identified by the argument **cond**. System resources allocated during initialization will be recollected by the system. For more information about thread synchronization and condition variable usage, refer to the AIX Operating System documentation.

Return Codes:

If errors are detected during the execution of this function, one of the following error codes will be returned:

EBUSY The condition variable **cond** is being used by another thread.

EINVAL The argument **cond** is invalid.

Example:

```
integer function f_pthread_cond_destroy(cond)
    type(f_pthread_cond_t), intent(inout):: cond
end function
```

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_cond_init

This function can be used to dynamically initialize a condition variable **cond**. Its attributes will be set according to the attribute object **cattr**, if it is provided; otherwise, its attributes will be set to the system default. After the condition variable is initialized successfully, it can be used to synchronize threads. For more information about thread synchronization and condition variable usage, refer to the AIX Operating System documentation.

Another method of initializing a condition variable is to initialize it statically using the Fortran constant **PTHREAD_COND_INITIALIZER**.

Return Codes:

If errors are detected during the execution of this function, one of the following error codes will be returned:

EBUSY The condition variable is already in use. It is initialized and not destroyed.

EINVAL The argument **cond** or **cattr** is invalid.

Example:

```
integer function f_pthread_cond_init(cond, cattr)
    type(f_pthread_cond_t), intent(out):: cond
    type(f_pthread_condattr_t), intent(in), optional:: cattr
end function
```

f_pthread_cond_signal

This function will unblock at least one thread waiting on the condition variable **cond**. If there is no thread waiting on this condition variable, the function will still succeed, but the next caller to **f_pthread_cond_wait** will be blocked, and will wait on the condition variable **cond**. For more information about thread synchronization and condition variable usage, refer to the AIX Operating System documentation.

Return Codes:

If errors are detected during the execution of this function, the following error code will be returned:

EINVAL The argument **cond** is invalid.

Example:

```
integer function f_pthread_cond_signal(cond)
    type(f_pthread_cond_t), intent(inout):: cond
end function
```

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures
f_pthread_cond_t A derived data type whose components are all private. Any object of this type should be manipulated through the appropriate interfaces provided in this module. In addition, objects of this type can be initialized at compile time using the Fortran constant **PTHREAD_COND_INITIALIZER**. This data type corresponds to the POSIX **pthread_cond_t**, which is the type of condition variable object.

f_pthread_cond_timedwait

This function can be used to wait for a certain condition to occur. The argument **mutex** must be locked before calling this function. The mutex is unlocked atomically and the calling thread waits for the condition to occur. The argument **timeout** specifies a deadline before which the condition must occur. If the deadline is reached before the condition occurs, the function will return an error code. This function provides a cancelation point in that the calling thread can be canceled if it is in the enabled state.

The argument **timeout** will specify an absolute date of the form: Oct. 31 10:00:53, 1998. For related information, see **f_maketime** and **f_timespec**. For information on the absolute date, refer to the AIX Operating System documentation.

Return Codes:

If errors are detected during the execution of this function, one of the following error codes is returned:

EINVAL The argument **cond**, **mutex**, or **timeout** is invalid.

EDEADLK The argument **mutex** is not locked by the calling thread.

ETIMEDOUT The waiting deadline was reached before the condition occurred.

Example:

```
integer function f_pthread_cond_timedwait(cond, mutex, timeout)
    type(f_pthread_cond_t), intent(inout):: cond
    type(f_pthread_mutex_t), intent(inout):: mutex
    type(f_timespec), intent(in):: timeout
end function
```

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_cond_wait

This function can be used to wait for a certain condition to occur. The argument **mutex** must be locked before calling this function. The mutex is unlocked atomically, and the calling thread waits for the condition to occur. If the condition does not occur, the function will wait until the calling thread is terminated in another way. This function provides a cancelation point in that the calling thread can be canceled if it is in the enabled state.

Return Codes:

When this function executes successfully, the mutex is locked again before the function returns. If errors are detected during the execution of this function, one of the following error codes will be returned:

EINVAL The argument **cond** or **mutex** is invalid.

EDEADLK The **mutex** is not locked by the calling thread.

Example:

```
integer function f_pthread_cond_wait(cond, mutex)
    type(f_pthread_cond_t), intent(inout):: cond
    type(f_pthread_mutex_t), intent(inout):: mutex
end function
```

f_pthread_condattr_destroy

This function can be called to destroy the condition variable attribute objects that are no longer required. The target object is identified by the argument **cattr**. The system resources allocated when it is initialized will be recollected.

Return Codes:

If errors are detected during the execution of this function, the following error code will be returned:

EINVAL The argument **cattr** is invalid.

Example:

```
integer function f_pthread_condattr_destroy(cattr)
    type(f_pthread_condattr_t), intent(inout):: cattr
end function
```

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_condattr_getpshared

This function can be used to query the process-shared attribute of the condition variable attributes object identified by the argument *cattr*. The current setting of this attribute will be returned in the argument *pshared*. *pshared* will contain one of the following values:

PTHREAD_PROCESS_SHARED The condition variable can be used by any thread that has access to the memory where it is allocated, even if these threads belong to different processes.

PTHREAD_PROCESS_PRIVATE The condition variable shall only be used by threads within the same process as the thread that created it.

Return Codes:

If this function completes successfully, value 0 is returned and the value of the process-shared attribute is returned through the argument *pshared*. Otherwise, the following error will be returned:

`EINVAL The argument cattr is invalid.`

Example:

```
integer(4) function f_pthread_condattr_getpshared(cattr, pshared)
type(f_pthread_condattr_t), intent(in):: cattr
integer(4), intent(out):: pshared
end function f_pthread_condattr_getpshared
```

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_condattr_setpshared

This function is used to set the process-shared attribute of the condition variable attributes object identified by the argument *cattr*. Its process-shared attribute will be set according to the argument *pshared*. *pshared* must have one of the following values:

PTHREAD_PROCESS_SHARED Specifies that the condition variable can be used by any thread that has access to the memory where it is allocated, even if these threads belong to different processes.

PTHREAD_PROCESS_PRIVATE Specifies that the condition variable shall only be used by threads within the same process as the thread that created it. This is the default setting of the attribute.

Return Codes:

If this function completes successfully, value 0 is returned. Otherwise, one of the following errors will be returned:

```
EINVAL  The argument cattr is invalid.
EINVAL  The value specified by the argument pshared
        is invalid.
```

Example:

```
integer(4) function f_pthread_condattr_setpshared(cattr, pshared)
type(f_pthread_condattr_t), intent(inout):: cattr
integer(4), intent(in):: pshared
end function f_pthread_condattr_setpshared
```

f_pthread_condattr_t

A derived data type whose components are all private. Any object of this type should be manipulated only through the appropriate interfaces provided in this module.

This data type corresponds to the POSIX **pthread_condattr_t**, which is the type of condition variable attribute object.

Functions, Subroutines, and Data Structures

f_pthread_create

This function is used to create a new thread in the current process. The newly created thread will assume the attributes defined in the thread attribute object **attr**, if it is provided. Otherwise, the new thread will have system default attributes. The new thread will begin execution at the subroutine **ent**, which is required to have one dummy argument. The system will pass the argument **arg** to the thread entry subroutine **ent** as its actual argument. The argument **flag**is used to inform the system of the property of the argument **arg**. When the execution returns from the entry subroutine **ent**, the new thread will terminate automatically.

If subroutine **ent** was declared such that an explicit interface would be required if it was called directly, then an explicit interface is also required when it is passed as an argument to this function.

Note that argument **arg** is a generic argument that can be of any type and any rank with the limitations detailed on page 727.

The argument **flag** must be used to convey the property of the argument **arg** exactly to the system.

The argument **flag** can assume a value which is one of, or a combination of, the following constants:

FLAG_CHARACTER:
: if the entry subroutine **ent** expects an argument of type **CHARACTER** in any way or any form, this flag value must be included to indicate this fact. However, if the subroutine expects a Fortran 90 pointer pointing to an argument of type **CHARACTER**, the **FLAG_DEFAULT** value should be included instead.

FLAG_ASSUMED_SHAPE:
: if the entry subroutine **ent** has a dummy argument which is an assumed-shape array of any rank, this flag value must be included to indicate this fact.

FLAG_DEFAULT:
: otherwise, this flag value is needed.

Return Codes:

If the call to this function is successful, the ID of the newly created thread will be returned through argument **thread**. Otherwise, one of the following error codes will be returned:

EAGAIN The system does not have enough resources to create a new thread.

EINVAL The argument **thread**, **attr**, or **flag** is invalid.

ENOMEM The system does not have sufficient memory to create a new thread.

Example:

```
integer function f_pthread_create(thread, attr, flag, ent, arg)
    type(f_pthread_t), intent(out):: thread
    type(f_pthread_attr_t), intent(in), optional:: attr
    integer(4), intent(in):: flag
    external ent
end function
```

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_detach
 	
This function is used to indicate to the pthreads library implementation that storage for the thread whose thread ID is specified by the argument thread can be claimed when this thread terminates. If the thread has not yet terminated, **f_pthread_detach** shall not cause it to terminate. Multiple **f_pthread_detach** calls on the same target thread cause an error.
 	
Return Codes:
 	
If this function completes successfully, value 0 is returned. Otherwise, the following error will be returned:
 	
`EINVAL The argument thread is invalid.`
 	
Example:
 	
`integer(4) function f_pthread_detach(thread)`
`type(f_pthread_t), intent(in):: thread`
`end function f_pthread_detach`

f_pthread_equal
 	
This function can be used to compare whether two thread ID's identify the same thread or not.
 	
Return Codes:
 	
TRUE The two thread ID's identify the same thread.
 	
FALSE The two thread ID's do not identify the same thread.
 	
Example:
 	
`logical function f_pthread_equal(thread1, thread2)`
`    type(f_pthread_t), intent(in):: thread1, thread2`
`end function`

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_exit

This subroutine can be called explicitly to terminate the calling thread before it returns from the entry subroutine. The actions taken depend on the state of the calling thread. If it is in non-detached state, the calling thread will wait to be joined. If the thread is in detached state, or when it is joined by another thread, the calling thread will terminate safely. First, the cleanup stack will be popped and executed, and then any thread-specific data will be destructed by the destructors. Finally, the thread resources are freed and the argument **ret** will be returned to the joining threads. The argument **ret** of this subroutine is optional. Currently, argument **ret** is limited to be an integer pointer. If it is not an integer pointer, the behavior is undefined.

Return Codes:

This subroutine never returns. If argument **ret** is not provided, NULL will be provided as this thread's exit status.

Example:

```
subroutine f_pthread_exit(ret)
    pointer(ret, byte)
    optional ret
    intent(in) ret
end subroutine
```

f_pthread_getconcurrency

This function returns the value of the concurrency level set by a previous call to the **f_pthread_setconcurrency** function. If the **f_pthread_setconcurrency** function was not previously called, this function returns zero to indicate that the system is maintaining the concurrency level. For more information about the concurrency level, refer to the AIX Operating System documentation.

Example:

```
integer(4) function f_pthread_getconcurrency()
end function f_pthread_getconcurrency
```

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_getschedparam

This function can be used to query the current setting of the scheduling property of the target thread. The target thread is identified by argument **thread**. Its scheduling policy will be returned through argument **policy** and its scheduling property through argument **param**. The sched_priority field in **param** defines the scheduling priority. The priority field will assume a value in the range of 1-127, where *127* is the most favored scheduling priority while *1* is the least.

Return Codes:

If errors are detected during the execution of this function, one of the following error codes will be returned:

ENOSYS The POSIX priority scheduling option is not implemented on AIX.

ESRCH The target thread does not exist.

Example:

```
integer function f_pthread_getschedparam(thread, policy, param)
    type(f_pthread_t), intent(in):: thread
    integer(4), intent(out):: policy
    type(f_sched_param), intent(out):: param
end function
```

f_pthread_getspecific

This function can be used to retrieve the thread-specific data associated with **key**. Note that the argument **arg** is not optional in this function as it will return the thread-specific data. If there is no data associated with the key for the calling thread, NULL will be returned. Currently, **arg** must be an integer pointer. If it is not an integer pointer, the result is undefined.

Return Codes:

If errors are detected during the execution of this function, the following error code will be returned:

EINVAL The argument **key** is invalid.

Example:

```
integer function f_pthread_getspecific(key, arg)
    type(f_pthread_key_t), intent(in):: key
    pointer(arg, byte)
    intent(out) arg
end function
```

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_join

This function can be called to join a particular thread designated by the argument **thread**. If the target thread is in non-detached state and is already terminated, this call will return immediately with the target thread's status returned in argument **ret** if it is provided. The argument **ret** is optional. Currently, **ret** must be an integer pointer if it is provided.

If the target thread is in detached state, it is an error to join it.

Return Codes:

If errors are detected during the execution of this function, one of the following error codes will be returned:

EDEADLK This call will cause a deadlock, or the calling thread is trying to join itself.

EINVAL The argument **thread** is invalid.

ESRCH The argument **thread** designates a thread which does not exist or is in detached state.

Example:

```
integer function f_pthread_join(thread, ret)
    type(f_pthread_t), intent(in):: thread
    optional ret
    intent(out) ret
    pointer(ret, byte)
end function
```

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_key_create

This function can be used to acquire a thread-specific data key. The key will be returned in the argument **key**. The argument **dtr** is a subroutine that will be used to destruct the thread-specific data associated with this key when any thread terminates after this calling point. The destructor will receive the thread-specific data as its argument. The destructor itself is optional. If it is not provided, the system will not invoke any destructor on the thread-specific data associated with this key. Note that the number of thread-specific data keys is limited in each process. It is the user's responsibility to manage the usage of the keys. The per-process limit can be checked by the Fortran constant **PTHREAD_DATAKEYS_MAX**.

Return Codes:

If errors are detected during the execution of this function, one of the following error codes will be returned:

EAGAIN The maximum number of keys has been exceeded.

EINVAL The argument **key** is invalid.

ENOMEM There is insufficient memory to create this key.

Example:

```
integer function f_pthread_key_create(key, dtr)
    type(f_pthread_key_t), intent(out):: key
    external dtr
    optional dtr
end function
```

f_pthread_key_delete

This function will destroy the thread-specific data key identified by the argument **key**. It is the user's responsibility to ensure that there is no thread-specific data associated with this key. This function does not call any destructor on the thread's behalf. After the key is destroyed, it can be reused by the system for **f_pthread_key_create** requests.

Return Codes:

If errors are detected during the execution of this function, one of the following error codes will be returned:

EINVAL The argument **key** is invalid.

EBUSY There is still data associated with this key.

Example:

```
integer function f_pthread_key_delete(key)
    type(f_pthread_key_t), intent(inout):: key
end function
```

Functions, Subroutines, and Data Structures

f_pthread_key_t

A derived data type whose components are all private. Any object of this type should be manipulated only through the appropriate interfaces provided in this module.

This data type corresponds to the POSIX **pthread_key_t**, which is the type of key object for accessing thread-specific data.

f_pthread_kill

This function can be used to send a signal to a target thread. The target thread is identified by argument **thread**. The signal which will be sent to the target thread is identified in argument **sig**. If **sig** contains value zero, error checking will be done by the system but no signal will be sent. For more information about signal management in multi-threaded systems, refer to the AIX Operating System documentation.

Return Codes:

If errors are detected during the execution of this function, one of the following error codes will be returned:

EINVAL The argument **thread** or **sig** is invalid.

ESRCH The target thread does not exist.

Example:

```
integer function f_pthread_kill(thread, sig)
    type(f_pthread_t), intent(inout):: thread
    integer(4), intent(in):: sig
end function
```

f_pthread_mutex_destroy

This function should be called to destroy those mutex objects that are no longer required. In this way, the system can recollect the memory resources. The target mutex object is identified by the argument **mutex**.

Return Codes:

If errors are detected during the execution of this function, one of the following error codes will be returned:

EBUSY The target mutex is locked or referenced by another thread.

EINVAL The argument **mutex** is invalid.

Example:

```
integer function f_pthread_mutex_destroy(mutex)
    type)f_pthread_mutex_t), intent(inout):: mutex
end function
```

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_mutex_getprioceiling

This function can be used to dynamically query the priority ceiling attribute of the mutex object identified by the argument **mutex**. The current ceiling value will be returned through the argument **old**.

Return Codes:

Note that this function cannot be used at present since mutex priority protection protocol is not implemented on AIX.

Example:

```
integer function f_pthread_mutex_getprioceiling(mutex, old)
    type(f_pthread_mutex_t), intent(in):: mutex
    integer(4), intent(out):: old
end function
```

f_pthread_mutex_init

This function can be used to initialize the mutex object identified by argument **mutex**. The initialized mutex will assume attributes set in the mutex attribute object **mattr**, if it is provided. If **mattr** is not provided, the system will initialize the mutex to have default attributes. After it is initialized, the mutex object can be used to synchronize accesses to critical data or code. It can also be used to build more complicated thread synchronization objects.

Another method to initialize mutex objects is to statically initialize them through the Fortran constant **PTHREAD_MUTEX_INITIALIZER**. If this method of initialization is used it is not necessary to call the function before using the mutex objects.

Return Codes:

If errors occur in the execution of this function, one of the following error codes will be returned:

EAGAIN The system did not have enough resources to initialize this mutex.

EBUSY This mutex is already in use. It was initialized and not destroyed.

EINVAL The argument **mutex** or **mattr** is invalid.

ENOMEM There is insufficient memory to initialize this mutex.

Example:

```
integer function f_pthread_mutex_init(mutex, mattr)
    type(f_pthread_mutex_t), intent(out):: mutex
    type(f_pthread_mutexattr_t), intent(in), optional:: mattr
end function
```

Functions, Subroutines, and Data Structures

f_pthread_mutex_lock

This function can be used to acquire ownership of the mutex object. (In other words, the function will lock the mutex.) If the mutex has already been locked by another thread, the caller will wait until the mutex is unlocked. If the mutex is already locked by the caller itself, an error will be returned to prevent recursive locking.

Return Codes:

If errors are detected during the execution of this function, one of the following error codes will be returned:

EDEADLK The mutex is locked by the calling thread already.

EINVAL The argument **mutex** is invalid.

Example:

```
integer function f_pthread_mutex_lock(mutex)
    type(f_pthread_mutex_t), intent(inout):: mutex
end function
```

f_pthread_mutex_setprioceiling

This function can be used to dynamically set the priority ceiling attribute of the mutex object identified by the argument **mutex**. The new ceiling will be set to the value contained in the argument **new**. The previous ceiling will be returned through the argument **old**. The argument **new** should assume an integer value with a range from 1 to 127.

Return Codes:

Note that this function cannot be used at present since mutex priority protection protocol is not implemented on AIX.

Example:

```
integer function f_pthread_mutex_setprioceiling(mutex, new, old)
    type(f_pthread_mutex_t), intent(inout):: mutex
    integer(4), intent(in):: new
    integer(4), intent(out):: old
end function
```

f_pthread_mutex_t

A derived data type whose components are all private. Any object of this type should be manipulated through the appropriate interfaces provided in this module. In addition, objects of this type can be initialized statically through the Fortran constant **PTHREAD_MUTEX_INITIALIZER**.

This data type corresponds to the POSIX **pthread_mutex_t**, which is the type of mutex object.

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_mutex_trylock

This function can be used to acquire ownership of the mutex object. (In other words, the function will lock the mutex.) If the mutex has already been locked by another thread, the function returns the error code **EBUSY**. The calling thread can check the return code to take further actions. If the mutex is already locked by the caller itself, an error will be returned to prevent recursive locking.

Return Codes:

If errors are detected during the execution of this function, one of the following error codes will be returned:

EBUSY The target mutex is locked or referenced by another thread.

EDEADLK The mutex is locked by the calling thread already.

EINVAL The argument **mutex** is invalid.

Example:

```
integer function f_pthread_mutex_trylock(mutex)
    type(f_pthread_mutex_t), intent(inout):: mutex
end function
```

f_pthread_mutex_unlock

This function should be called to release the mutex object's ownership as soon as possible in order to allow other threads to lock the mutex.

Return Codes:

If errors are detected during the execution of this function, one of the following error codes will be returned:

EINVAL The argument **mutex** is invalid.

EPERM The mutex is not locked by the calling thread.

Example:

```
integer function f_pthread_mutex_unlock(mutex)
    type(f_pthread_mutex_t), intent(inout):: mutex
end function
```

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_mutexattr_destroy

This function can be used to destroy a mutex attribute object that has been initialized previously. Allocated memory will then be recollected. A mutex created with this attribute will not be affected by this action.

Return Codes:

If errors are detected during the execution of this function, the following error code will be returned:

EINVAL The argument **mattr** is invalid.

Example:

```
integer function f_pthread_mutexattr_destroy(mattr)
    type(f_pthread_mutexattr_t), intent(inout):: mattr
end function
```

f_pthread_mutexattr_getprioceiling

This function can be used to query the mutex priority ceiling attribute in the mutex attribute object identified by argument **mattr**. The ceiling attribute will be returned through argument **ceiling**.

Return Codes:

Note that this function cannot be used at present since the mutex priority protection protocol is not implemented on AIX.

Example:

```
integer function f_pthread_mutexattr_getprioceiling(mattr, ceiling)
    type(f_pthread_mutexattr_t), intent(in):: mattr
    integer(4), intent(out):: ceiling
end function
```

f_pthread_mutexattr_getprotocol

This function can be used to query the current setting of mutex protocol attribute in the mutex attribute object identified by argument **mattr**. The protocol attribute will be returned through argument **proto**.

Return Codes:

Note that this function cannot be used at present since mutex priority inheritance and priority protection are not implemented on AIX.

Example:

```
integer function f_pthread_mutexattr_getprotocol(mattr, proto)
    type(f_pthread_mutexattr_t), intent(in):: mattr
    integer(4), intent(out):: proto
end function
```

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_mutexattr_getpshared

This function is used to query the process-shared attribute in the mutex attributes object identified by the argument *mattr*. The current setting of the attribute will be returned through the argument *pshared*. *pshared* will contain one of the following values:

PTHREAD_PROCESS_SHARED The mutex can be operated upon by any thread that has access to the memory where the mutex is allocated, even if the mutex is allocated in memory that is shared by multiple processes.

PTHREAD_PROCESS_PRIVATE The mutex will only be operated upon by threads created within the same process as the thread that initialized the mutex.

Return Codes:

If this function completes successfully, value 0 is returned and the value of the process-shared attribute is returned through the argument *pshared*. Otherwise, the following error will be returned:

```
EINVAL The argument mattr is invalid.
```

Example:

```
integer(4) function f_pthread_mutexattr_getpshared(mattr, pshared)
type(f_pthread_mutexattr_t), intent(in):: mattr
integer(4), intent(out):: pshared
end function f_pthread_mutexattr_getpshared
```

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_mutexattr_gettype
 	
This function is used to query the mutex type attribute in the mutex attributes object identified by the argument*mattr*.
 	
If this function completes successfully, value 0 is returned and the type attribute will be returned through the argument type. The argument type will contain one of the following values:
 	
PTHREAD_MUTEX_NORMAL This type of mutex does not detect deadlock. A thread attempting to relock this mutex without first unlocking it will deadlock. Attempting to unlock a mutex locked by a different thread results in undefined behavior.
 	
PTHREAD_MUTEX_ERRORCHECK This type of mutex provides error checking. A thread attempting to relock this mutex without first unlocking it will return with an error. A thread attempting to unlock a mutex which another thread has locked will return an error. A thread attempting to unlock an unlocked mutex will return with an error.
 	
PTHREAD_MUTEX_RECURSIVE A thread attempting to relock this mutex without first unlocking it will succeed in locking the mutex. The relocking deadlock that can occur with mutexes of type **PTHREAD_MUTEX_NORMAL** cannot occur with this type of mutex. Multiple locks of this mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex.
 	
Return Codes:
 	
If this function fails, the following error will be returned:
 	
`EINVAL  The argument mattr is invalid.`
 	
Example:
 	
`integer(4) function f_pthread_mutexattr_gettype(mattr, type)`
`type(f_pthread_mutexattr_t), intent(in):: mattr`
`integer(4), intent(out):: type`
`end function f_pthread_mutexattr_gettype`

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_mutexattr_init

This function can be used to initialize a mutex attribute object before it can be used in any other way. The mutex attribute object will be returned through argument **mattr**.

Return Codes:

If errors are detected during the execution of this function, one of the following error codes will be returned:

EINVAL The argument **mattr** is invalid.

ENOMEM There is insufficient memory to create the object.

Example:

```
integer function f_pthread_mutexattr_init(mattr)
    type(f_pthread_mutexattr_t), intent(out):: mattr
end function
```

f_pthread_mutexattr_setprioceiling

This function can be used to set the mutex priority ceiling attribute in the mutex attribute object identified by the argument **mattr**. Argument **ceiling** is an integer with a range from 1 to 127. This attribute has an effect only if the mutex priority protection protocol is used.

Return Codes:

Note that this function cannot be used at present since the priority protection protocol is not implemented on AIX.

Example:

```
integer function f_pthread_mutexattr_setprioceiling(mattr, ceiling)
    type(f_pthread_mutexattr_t), intent(inout):: mattr
    integer(4), intent(in):: ceiling
end function
```

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_mutexattr_setprotocol

This function can be used to set the mutex protocol attribute in the mutex attribute object identified by argument **mattr**. Argument **proto** identifies the mutex protocol to be set. For more information about the set of valid values for **proto**, refer to the AIX Operating System documentation.

Return Codes:

Note that this function cannot be used at present since mutex priority inheritance and priority protection are not implemented on AIX.

Example:

```
integer function f_pthread_mutexattr_setprotocol(mattr, proto)
    type(p_thread_mutexattr_t), intent(inout):: mattr
    integer(4), intent(in):: proto
end function
```

f_pthread_mutexattr_setpshared

This function is used to set the process-shared attribute of the mutex attributes object identified by the argument *mattr*. The argument *pshared* must have one of the following values:

PTHREAD_PROCESS_SHARED Specifies the mutex can be operated upon by any thread that has access to the memory where the mutex is allocated, even if the mutex is allocated in memory that is shared by multiple processes.

PTHREAD_PROCESS_PRIVATE Specifies the mutex will only be operated upon by threads created within the same process as the thread that initialized the mutex. This is the default setting of the attribute.

Return Codes:

If this function completes successfully, value 0 is returned. Otherwise, one of the following errors will be returned:

```
EINVAL  The argument mattr is invalid.
EINVAL  The value specified by the argument pshared
         is invalid.
```

Example:

```
integer(4) function f_pthread_mutexattr_setpshared(mattr, pshared)
type(f_pthread_mutexattr_t), intent(inout):: mattr
integer(4), intent(in):: pshared
end function f_pthread_mutexattr_setpshared
```

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_mutexattr_settype

This function is used to set the mutex type attribute in the mutex attributes object identified by the argument *mattr* The argument type identifies the mutex type attribute to be set. For more information about the type of a mutex, refer to the AIX Operating System documentation.

The argument type must contain one of the following values:

PTHREAD_MUTEX_NORMAL
This type of mutex does not detect deadlock. A thread attempting to relock this mutex without first unlocking it will deadlock. Attempting to unlock a mutex locked by a different thread results in undefined behavior.

PTHREAD_MUTEX_ERRORCHECK
This type of mutex provides error checking. A thread attempting to relock this mutex without first unlocking it will return with an error. A thread attempting to unlock a mutex which another thread has locked will return an error. A thread attempting to unlock an unlocked mutex will return with an error.

PTHREAD_MUTEX_RECURSIVE
A thread attempting to relock this mutex without first unlocking it will succeed in locking the mutex. The relocking deadlock that can occur with mutexes of type **PTHREAD_MUTEX_NORMAL** cannot occur with this type of mutex. Multiple locks of this mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex.

PTHREAD_MUTEX_DEFAULT
The same as **PTHREAD_MUTEX_NORMAL.**

Note: The behavior of AIX 4.1 and AIX 4.2 mutexes is similar to the type **PTHREAD_MUTEX_ERRORCHECK** in AIX 4.3.

Return Codes:

If this function completes successfully, value 0 is returned. Otherwise, one of the following errors will be returned:

```
EINVAL  The value of type is invalid.
EINVAL  The argument mattr is invalid.
```

Example:

```
integer(4) function f_pthread_mutexattr_settype(mattr, type)
type(f_pthread_mutexattr_t), intent(inout):: mattr
integer(4), intent(in):: type
end function f_pthread_mutexattr_settype
```

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_mutexattr_t

A derived data type whose components are all private. Any object of this type should be manipulated only through the appropriate interfaces provided in this module.

This data type corresponds to the POSIX **pthread_mutexattr_t**, which is the type of mutex attribute object.

f_pthread_once

This function can be used to initialize those data required to be initialized only once. The first thread calling this function will call **initr** to do the initialization. Other threads calling this function afterwards will have no effect. Argument **initr** must be a subroutine without dummy arguments.

Return Codes:

If errors are detected during the execution of this function, the following error code will be returned:

EINVAL The argument **once** or **initr** is invalid.

Example:

```
integer function f_pthread_once(once, initr)
    type(f_pthread_once_t), intent(inout):: once
    external initr
end function
```

f_pthread_once_t

A derived data type whose components are all private. Any object of this type should be manipulated through the appropriate interfaces provided in this module. However, objects of this type can *only* be initialized through the Fortran constant **PTHREAD_ONCE_INIT**.

This data type corresponds to the POSIX **pthread_once_t**, which is the type of once-block object.

f_pthread_rwlock_destroy

This function destroys the read-write lock object specified by the argument *rwlock* and releases any resources used by the lock.

Return Codes:

If this function completes successfully, value 0 is returned. Otherwise, one of the following errors will be returned:

```
EBUSY   The target read-write lock object is locked.
EINVAL  The argument rwlock is invalid.
```

Example:

```
integer(4) function f_pthread_rwlock_destroy(rwlock)
type(f_pthread_rwlock_t), intent(inout):: rwlock
end function f_pthread_rwlock_destroy
```

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_rwlock_init

This function initializes the read-write lock object specified by *rwlock* with the attribute specified by the argument *rwattr*. If the optional argument *rwattr* is not provided, the system will initialize the read-write lock object with the default attributes. After it is initialized, the lock can be used to synchronize access to critical data. With a read-write lock, many threads can have simultaneous read-only access to data, while only one thread can have write access at any given time and no other readers or writers are allowed. For more information about thread synchronization and read-write lock usage, refer to the AIX Operating System documentation.

Another method to initialize read-write lock objects is to statically initialize them through the Fortran constant **PTHREAD_RWLOCK_INITIALIZER.** If this method of initialization is used, it is not necessary to call this function before using the read-write lock objects.

Return Codes:

If this function completes successfully, value 0 is returned. Otherwise, one of the following errors will be returned:

```
EAGAIN  The system did not have enough resources to
        initialize this read-write lock.
ENOMEM  There is insufficient memory to initialize
        this read-write lock.
EBUSY   This read-write lock is already in use. It
        was initialized and not yet destroyed.
EINVAL  The argument rwlock or rwattr is invalid.
EPERM   The caller does not have privilege to perform
        the operation.
```

Example:

```
integer(4) function f_pthread_rwlock_init(rwlock, rwattr)
type(f_pthread_rwlock_t), intent(out):: rwlock
type(f_pthread_rwlockattr_t), intent(in), optional:: rwattr
end function f_pthread_rwlock_init
```

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_rwlock_rdlock
 	
This function applies a read lock to the read-write lock specified by the argument *rwlock*. The calling thread acquires the read lock if a writer does not hold the lock and there are no writes blocked on the lock. If a writer holds the lock, the calling thread will not acquire the read lock. If the read lock is not acquired, the calling thread blocks (that is, it does not return from the **f_pthread_rwlock_rdlock** call) until it can acquire the lock. Results are undefined if the calling thread holds a write lock on *rwlock* at the time the call is made. A thread may hold multiple concurrent read locks on *rwlock* (that is, successfully call the **f_pthread_rwlock_rdlock** function *n* times). If so, the thread must perform matching unlocks (that is, it must call the **f_pthread_rwlock_unlock** function *n* times).
 	
Return Codes:
 	
If this function completes successfully, value 0 is returned. Otherwise, one of the following errors will be returned:
`EINVAL  The argument rwlock does not refer to an`
`        initialized read-write lock object.`
`EAGAIN  The read-write lock could not be acquired`
`        because the maximum number of read`
`        locks for rwlock has been exceeded.`
 	
Example:
`integer(4) function f_pthread_rwlock_rdlock(rwlock)`
`type(f_pthread_rwlock_t), intent(inout):: rwlock`
`end function f_pthread_rwlock_rdlock`

f_pthread_rwlock_t A derived data type whose components are all private. Any object of this type should be manipulated only through the appropriate interfaces provided in this module. In addition, objects of this type can be initialized statically through the Fortran constant **PTHREAD_RWLOCK_INITIALIZER**. This data type corresponds to the AIX 4.3 data type **pthread_rwlock_t**, which is the type of the read-write lock objects.

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_rwlock_tryrdlock

This function applies a read lock like the **f_pthread_rwlock_rdlock** function with the exception that the function fails if any thread holds a write lock on *rwlock* or there are writers blocked on *rwlock*. In that case, the function returns EBUSY. The calling thread can check the return code to take further actions.

Return Codes:

This function returns zero if the lock for reading on the read-write lock object specified by *rwlock* is acquired. Otherwise, one of the following errors will be returned:

```
EINVAL  The argument rwlock does not refer to an
        initialized read-write lock object.
EBUSY   The read-write lock could not be acquired
        for reading because a writer holds the lock
        or was blocked on it.
EAGAIN  The read-write lock could not be acquired
        because the maximum number of
        read locks for rwlock has been exceeded.
EDEADLK The current thread already owns the
        read-write lock for writing.
```

Example:

```
integer(4) function f_pthread_rwlock_tryrdlock(rwlock)
type(f_pthread_rwlock_t), intent(inout):: rwlock
end function f_pthread_rwlock_tryrdlock
```

f_pthread_rwlock_trywrlock

This function applies a write lock like the **f_pthread_rwlock_wrlock** function with the exception that the function fails if any thread currently holds *rwlock* (for reading or writing). In that case, the function returns EBUSY. The calling thread can check the return code to take further actions.

Return Codes:

This function returns zero if the lock for writing on the read-write lock object specified by *rwlock* is acquired. Otherwise, one of the following errors will be returned:

```
EINVAL  The argument rwlock does not refer to an
        initialized read-write lock object.
EBUSY   The read-write lock could not be acquired for
        writing because it was already locked for
        reading or writing.
EDEADLK The current thread already owns the read-write
        lock for writing or reading.
```

Example:

```
integer(4) function f_pthread_rwlock_trywrlock(rwlock)
type(f_pthread_rwlock_t), intent(inout):: rwlock
end function f_pthread_rwlock_trywrlock
```

Functions, Subroutines, and Data Structures

f_pthread_rwlock_unlock

This function is used to release a lock held on the read-write lock object specified by the argument *rwlock*. If this function is called to release a read lock from the read-write lock object and there are other read locks currently held on this read-write lock object, the read-write lock object remains in the read locked state. If this function releases the calling thread's last read lock on this read-write lock object, then the calling thread is no longer one of the owners of the object. If this function releases the last read lock for this read-write lock object, the read-write lock object will be put in the unlocked state with no owners.

Return Codes:

If this function completes successfully, value 0 is returned. Otherwise, one of the following errors will be returned:

```
EINVAL  The argument rwlock does not refer to an
        initialized read-write lock object.
EPERM   The current thread does not own the read-write
        lock.
```

Example:

```
integer(4) function f_pthread_rwlock_unlock(rwlock)
type(f_pthread_rwlock_t), intent(inout):: rwlock
end function f_pthread_rwlock_unlock
```

f_pthread_rwlock_wrlock

This function applies a write lock to the read-write lock specified by the argument *rwlock*. The calling thread acquires the write lock if no other thread (reader or writer) holds the read-write lock *rwlock*. Otherwise, the thread blocks (that is, does not return from the **f_pthread_rwlock_wrlock** call) until it acquires the lock. Results are undefined if the calling thread holds the read-write lock (whether a read or write lock) at the time the call is made.

Return Codes:

If this function completes successfully, value 0 is returned. Otherwise, the following error will be returned:

```
EINVAL The argument rwlock does not refer to an
       initialized read-write lock object.
```

Example:

```
integer(4) function f_pthread_rwlock_wrlock(rwlock)
type(f_pthread_rwlock_t), intent(inout):: rwlock
end function f_pthread_rwlock_wrlock
```

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_rwlockattr_destroy

This function destroys a read-write lock attributes object specified by the argument *rwattr* which has been initialized previously. A read-write lock created with this attribute will not be affected by the action.

Return Codes:

If this function completes successfully, value 0 is returned. Otherwise, the following error will be returned:

```
EINVAL  The argument rwattr is invalid.
```

Example:

```
integer(4) function f_pthread_rwlockattr_destroy(rwattr)
type(f_pthread_rwlockattr_t), intent(inout):: rwattr
end function f_pthread_rwlockattr_destroy
```

f_pthread_rwlockattr_getpshared

This function is used to obtain the value of the process-shared attribute from the initialized read-write lock attributes object specified by the argument *rwattr*. The current setting of this attribute will be returned in the argument*pshared*. *pshared* will contain one of the following values:

PTHREAD_PROCESS_SHARED The read-write lock can be operated upon by any thread that has access to the memory where it is allocated, even if these threads belong to different processes.

PTHREAD_PROCESS_PRIVATE The read-write lock shall only be used by threads within the same process as the thread that created it.

Return Codes:

If this function completes successfully, value 0 is returned and the value of the process-shared attribute of *rwattr* is stored into the object specified by the argument *pshared*. Otherwise, the following error will be returned:

```
EINVAL  The argument rwattr is invalid.
```

Example:

```
integer(4) function f_pthread_rwlockattr_getpshared(rwattr, pshared)
type(f_pthread_rwlockattr_t), intent(in):: rwattr
integer(4), intent(out):: pshared
end function f_pthread_rwlockattr_getpshared
```

Functions, Subroutines, and Data Structures

f_pthread_rwlockattr_init
 	
This function initializes a read-write lock attributes object specified by *rwattr* with the default value for all of the attributes.
 	
Return Codes:
 	
If this function completes successfully, value 0 is returned. Otherwise, the following error will be returned:
 	
`ENOMEM  There is insufficient memory to initialize the read-write lock attributes object.`
 	
Example:
 	
`integer(4) function f_pthread_rwlockattr_init(rwattr)`
`type(f_pthread_rwlockattr_t), intent(out):: rwattr`
`end function f_pthread_rwlockattr_init`

f_pthread_rwlockattr_setpshared
 	
This function is used to set the process-shared attribute in an initialized read-write lock attributes object specified by the argument *rwattr*. The argument *pshared* must have one of the following values:
 	
PTHREAD_PROCESS_SHARED Specifies the read-write lock can be operated upon by any thread that has access to the memory where it is allocated, even if these threads belong to different processes.
 	
PTHREAD_PROCESS_PRIVATE Specifies the read-write lock shall only be used by threads within the same process as the thread that created it. This is the default setting of the attribute.
 	
Return Codes:
 	
If this function completes successfully, value 0 is returned. Otherwise, one of the following errors will be returned:
 	
`EINVAL  The argument rwattr is invalid.`
`EINVAL  The new value specified for the attribute is invalid.`
 	
Example:
 	
`integer(4) function f_pthread_rwlockattr_setpshared(rwattr, pshared)`
`type(f_pthread_rwlockattr_t), intent(inout):: rwattr`
`integer(4), intent(in):: pshared`
`end function f_pthread_rwlockattr_setpshared`

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_rwlockattr_t

This is a derived data type whose components are all private. Any object of this type should be manipulated only through the appropriate interfaces provided in this module.

This data type corresponds to the AIX 4.3 data type **pthread_rwlockattr_t**, which is the type of the read-write lock attributes objects.

---

f_pthread_self

This function can be used to return the thread ID of the calling thread.

Example:

```
type(f_pthread_t) function f_pthread_self()
end function
```

Return Codes:

The calling thread's ID is returned.

---

f_pthread_setcancelstate

This function can be used to set the thread's cancelability state. The new state will be set according to the argument **state**. The old state will be returned in the argument **oldstate**. These arguments will assume the value of one of the following Fortran constants:

PTHREAD_CANCEL_DISABLE: the thread's cancelability is disabled.

PTHREAD_CANCEL_ENABLE: the thread's cancelability is enabled.

Return Codes:

If errors are detected during the execution of this function, the following error code will be returned:

EINVAL The argument **state** is invalid.

Example:

```
integer function f_pthread_setcancelstate(state, oldstate)
    integer(4), intent(in):: state
    integer(4), intent(out):: oldstate
end function
```

Table 18. Fortran Pthreads Library Module (continued)

Functions, Subroutines, and Data Structures

f_pthread_setcanceltype

This function can be used to set the thread's cancelability type. The new type will be set according to the argument **type**. The old type will be returned in argument **oldtype**. These arguments will assume the value of one of the following Fortran constants:

PTHREAD_CANCEL_DEFERRED:
cancelation request will be delayed until a cancelation point.

PTHREAD_CANCEL_ASYNCHRONOUS:
cancelation request will be acted upon immediately. This may cause unexpected results.

Return Codes:

If errors are detected during the execution of this function, the following error code will be returned:

EINVAL The argument **type** is invalid.

Example:

```
integer function f_pthread_setcanceltype(type, oldtype)
    integer(4), intent(in):: type
    integer(4), intent(out):: oldtype
end function
```

f_pthread_setconcurrency

This function is used to inform the pthreads library implementation of desired concurrency level as specified by the argument *new_level*. The actual level of concurrency provided by the implementation as a result of this function call is unspecified. For more information about the concurrency level, refer to the AIX Operating System documentation.

Return Codes:

If this function completes successfully, value 0 is returned. Otherwise, one of the following errors will be returned:

```
EINVAL The value specified by new_level is negative.
EAGAIN The value specified by new_level would cause
       system resource to be exceeded.
```

Example:

```
integer(4) function f_pthread_setconcurrency(new_level)
integer(4), intent(in):: new_level
end function f_pthread_setconcurrency
```

Functions, Subroutines, and Data Structures

f_pthread_setschedparam

This function can be used to dynamically set the scheduling policy and the scheduling property of a thread. The target thread is identified by argument **thread**. The new scheduling policy for the target thread is provided through argument **policy**. The valid scheduling policies on AIX can be found in the AIX Operating System documentation. The new scheduling property of the target thread will be set to the value provided by argument **param**. The sched_priority field in **param** defines the scheduling priority. Its range is 1-127.

The new policy cannot be set to first-in first-out or round-robin unless the caller has root authority. For more details about when the new scheduling property has effect on the target thread, refer to the AIX Operating System documentation.

Return Codes:

If errors are detected during the execution of this function, one of the following error codes will be returned:

EINVAL
The argument **thread** or **param** is invalid.

ENOSYS
The POSIX priority scheduling option is not implemented on AIX.

ENOTSUP
The value of argument **policy** or **param** is not supported.

EPERM
The target thread is not permitted to perform the operation or is in a mutex protocol already.

ESRCH
The target thread does not exist.

Example:

```
integer function f_pthread_setschedparam(thread, policy, param)
    type(f_pthread_t), intent(inout):: thread
    integer(4), intent(in):: policy
    type(f_sched_param), intent(in):: param
end function
```

Functions, Subroutines, and Data Structures

f_pthread_setspecific

This function can be used to set the calling thread's specific data associated with the key identified by argument **key**. The argument **arg**, which is optional, identifies the thread-specific data to be set. If **arg** is not provided, the thread-specific data will be set to NULL, which is the initial value for each thread. Currently, only an integer pointer can be passed as the **arg** argument. If **arg** is not an integer pointer, the result is undefined.

RETURN CODES:

If errors are detected during the execution of this function, one of the following error codes will be returned:

EINVAL The argument **key** is invalid.

ENOMEM There is insufficient memory to associate the data with the key.

Example:

```
integer function f_pthread_setspecific(key, arg)
    type(f_pthread_key_t), intent(in):: key
    pointer(arg, byte)
    optional arg
    intent(in) arg
end function
```

f_pthread_t

A derived data type whose components are all private. Any object of this type should be manipulated only through the appropriate interfaces provided in this module.

This data type corresponds to the POSIX **pthread_t**, which is the type of thread object.

f_pthread_testcancel

This subroutine provides a cancelation point in a thread. When it is called, any pending cancelation request will be acted upon immediately if it is in the enabled state.

Example:

```
subroutine f_pthread_testcancel()
end subroutine
```

Functions, Subroutines, and Data Structures

---

f_sched_param

This data type corresponds to the AIX system data structure **sched_param**, which is a system data type. See AIX Operating System documentation for more information.

Example:

This is a public data structure defined as:

```
type f_sched_param
    sequence
    integer sched_priority
    integer sched_policy
    integer reserved(6)
end type f_sched_param
```

---

f_sched_yield

This function is used to force the calling thread to relinquish the processor until it again becomes the head of its thread list.

Return Codes:

If this function completes successfully, value 0 is returned. Otherwise, a value of -1 will be returned.

Example:

```
integer(4) function f_sched_yield()
end function f_sched_yield
```

---

f_timespec

This is a Fortran definition of the AIX system data structure **timespec**. Within the Fortran Pthreads module, objects of this type are used to specify an absolute date and time. This *deadline absolute date* is used when waiting on a POSIX condition variable. See AIX Operating System documentation for more information.

Example:

This is a public data structure defined as:

```
type f_timespec
    sequence
    integer tv_sec
    integer tv_nsec
end type f_timespec
```

Limitation and Caveats on the Use of the Argument arg

Keep the following points in mind when you use the Argument **arg**:

1. Array sections with vector subscripts are not supported. They should not be passed to **arg**; otherwise, the result is unpredictable.
2. If the actual argument **arg** is an array section, the corresponding dummy argument in subroutine **cleanup** (for **f_pthread_cleanup_push**), or **ent** (for **f_pthread_create**), must be an assumed-shape array. Otherwise, the result is unpredictable.
3. If the actual argument **arg** has the pointer attribute that points to an array or array section, the corresponding dummy argument in subroutine **cleanup** (for **f_pthread_cleanup_push**), or **ent** (for **f_pthread_create**), must be a Fortran 90 pointer attribute or an assumed-shape array. Otherwise, the result is unpredictable.
4. The actual argument **arg** must be a variable. In other words, the actual argument must be eligible as a left-value in an assignment statement.

Part 5. Appendixes

Appendix A. Compatibility Across Standards

This information is provided for the benefit of FORTRAN 77 users who are unfamiliar with Fortran 95, Fortran 90 and XL Fortran.

Except as noted here, the Fortran 90 and Fortran 95 standards are upward-compatible extensions to the preceding Fortran International Standard, ISO 1539-1:1980, informally referred to as FORTRAN 77. Any standard-conforming FORTRAN 77 program remains standard-conforming under the Fortran 90 standard, except as noted under item 4 below regarding intrinsic procedures. Any standard-conforming FORTRAN 77 program remains standard-conforming under the Fortran 95 standard as long as none of the deleted features are used in the program, except as noted under item 4 below regarding intrinsic procedures. The Fortran 90 and Fortran 95 standard restricts the behavior of some features that are processor-dependent in FORTRAN 77. Therefore, a standard-conforming FORTRAN 77 program that uses one of these processor-dependent features may have a different interpretation under the Fortran 90 or Fortran 95 standard, yet remain a standard-conforming program. The following FORTRAN 77 features have different interpretations in Fortran 90 and Fortran 95:

1. FORTRAN 77 permitted a processor to supply more precision derived from a real constant than can be contained in a real datum when the constant is used to initialize a **DOUBLE PRECISION** data object in a **DATA** statement. Fortran 90 and Fortran 95 do not permit a processor this option.

   Previous releases of XL Fortran have been consistent with the Fortran 90 and Fortran 95 behavior.
2. If a named variable that is not in a common block is initialized in a **DATA** statement and does not have the **SAVE** attribute specified, FORTRAN 77 left its **SAVE** attribute processor-dependent. The Fortran 90 and Fortan 95 standards specify that this named variable has the **SAVE** attribute.

   Previous releases of XL Fortran have been consistent with the Fortran 90 and Fortran 95 behavior.
3. FORTRAN 77 required that the number of characters required by the input list must be less than or equal to the number of characters in the record during formatted input. The Fortran 90 and Fortran 95 standards specify that the input record is logically padded with blanks if there are not enough characters in the record, unless the **PAD='NO'** specifier is indicated in an appropriate **OPEN** statement.

   With XL Fortran, the input record is not padded with blanks if the **noblankpad** suboption of the **-qxlf77** compiler option is specified.

© Copyright IBM Corp. 1990, 2000

4. The Fortran 90 and Fortan 95 standards have more intrinsic functions than FORTRAN 77, in addition to a few intrinsic subroutines. Therefore, a standard-conforming FORTRAN 77 program may have a different interpretation under Fortran 90 and Fortran 95 if it invokes a procedure having the same name as one of the new standard intrinsic procedures, unless that procedure is specified in an **EXTERNAL** statement.

   With XL Fortran, the **-qextern** compiler option also treats specified names as if they appear in an **EXTERNAL** statement.
5. In Fortran 95, for some edit descriptors a value of 0 for a list item in a formatted output statement will be formatted differently. In addition, the Fortran 95 standard unlike the FORTRAN 77 standard specifies how rounding of values will affect the output field form. Therefore, for certain combinations of values and edit descriptors FORTRAN 77 processors may produce a different output form than Fortran 95 processors.
6. Fortran 95 allows a processor to distinguish between a positive and a negative real zero, whereas Fortran 90 did not. Fortran 95 changes the behavior of the **SIGN** intrinsic function when the second argument is negative real zero.

Fortran 90 compatibility

Except as noted here, the Fortran 95 standard is an upward-compatible extension to the preceding Fortran International Standard, ISO/IEC 1539-1:1991, informally referred to as Fortran 90. A standard conforming Fortran 90 program that does not use any of the features deleted from the Fortran 95 standard, is a standard conforming Fortran 95 program, as well. The Fortran 90 features that have been deleted from the Fortran 95 standard are the following:

- **ASSIGN** and assigned **GO TO** statements
- **PAUSE** statement
- **DO** control variables and expressions of type real
- **H** edit descriptor
- Branching to an **END IF** statement from outside the **IF** block

Fortran 95 allows a processor to distinguish between a positive and a negative real zero, whereas Fortran 90 did not. Fortran 95 changes the behavior of the **SIGN** intrinsic function when the second argument is negative real zero.

More intrinsic functions appear in the Fortran 95 standard than in the Fortran 90 standard. Therefore, a program that conforms to the Fortran 90 standard may have a different interpretation under the Fortran 95 standard. The different interpretation of the program in Fortran 95 will only occur if the program invokes a procedure that has the same name as one of the new

standard intrinsic procedures, unless that procedure is specified in an **EXTERNAL** statement or with an interface body.

Obsolescent Features

As the Fortran language evolves, it is only natural that the functionality of some older features are better handled by newer features geared toward today's programming needs. At the same time, the considerable investment in legacy Fortran code suggests that it would be insensitive to customer needs to decommit any Fortran 90 or FORTRAN 77 features at this time. For this reason, XL Fortran is fully upward compatible with the Fortran 90 and FORTRAN 77 standards. Fortran 95 has removed features that were part of both the Fortran 90 and FORTRAN 77 language standards. However, functionality has not been removed from Fortran 95 as efficient alternatives to the features deleted do exist.

Fortran 95 defines two categories of outmoded features: deleted features and obsolescent features. Deleted features are Fortran 90 or FORTRAN 77 features that are considered to be largely unused and so are not supported in Fortan 95.

Obsolescent features are FORTRAN 77 features that are still frequently used today but whose use can be better delivered by newer features and methods. Although obsolescent features are, by definition, supported in the Fortran 95 standard, some of them may be marked as deleted in the next Fortran standard. Although a processor may still support deleted features as extensions to the language, you may want to take steps now to modify your existing code to use better methods.

Fortran 90 indicates the following FORTRAN 77 features are obsolescent:

- Arithmetic **IF**

  Recommended method: Use the logical **IF** statement, **IF** construct, or **CASE** construct.
- **DO** control variables and expressions of type real

  Recommended method: Use variables and expression of type integer.
- **PAUSE** statement

  Recommended method: Use the **READ** statement.
- Alternate return specifiers

  Recommended method: Evaluate a return code in a **CASE** construct or a computed **GO TO** statement on return from the procedure.

  ```
  ! FORTRAN 77

    CALL SUB(A,B,C,*10,*20,*30)
  ```

```
! Fortran 90

  CALL SUB(A,B,C,RET_CODE)
  SELECT CASE (RET_CODE)
    CASE (1)
    ⋮
    CASE (2)
    ⋮
    CASE (3)
    ⋮
  END SELECT
```

- **ASSIGN** and assigned **GO TO** statements

  Recommended method: Use internal procedures.
- Branching to an **END IF** statement from outside the **IF** block

  Recommended method: Branch to the statement that follows the **END IF** statement.
- Shared loop termination and termination on a statement other than **END DO** or **CONTINUE**

  Recommended method: Use an **END DO** or **CONTINUE** statement to terminate each loop.
- **H** edit descriptor

  Recommended method: Use the character constant edit descriptor.

Fortran 95 indicates the following FORTRAN 77 features as obsolescent:

- Arithmetic **IF**

  Recommended method: Use the logical **IF** statement, **IF** construct, or **CASE** construct.
- Alternate return specifiers

  Recommended method: Evaluate a return code in a **CASE** construct or a computed **GO TO** statement on return from the procedure.

```
! FORTRAN 77

  CALL SUB(A,B,C,*10,*20,*30)

! Fortran 90

  CALL SUB(A,B,C,RET_CODE)
  SELECT CASE (RET_CODE)
    CASE (1)
    ⋮
```

```
  CASE (2)
 ⋮
  CASE (3)
 ⋮
END SELECT
```

- Shared loop termination and termination on a statement other than **END DO** or **CONTINUE**

  Recommended method: Use an **END DO** or **CONTINUE** statement to terminate each loop.
- Statement functions
- **DATA** statements in executables
- Assumed length character functions
- Fixed source form
- **CHARACTER*** form of declaration

Deleted Features

Fortran 95 indicates that the following Fortran 90 and FORTRAN 77 features have been deleted:

- **ASSIGN** and assigned **GO TO** statements
- **PAUSE** statement
- **DO** control variables and expressions of type real
- **H** edit descriptor
- Branching to an **END IF** statement from outside the **IF** block

Appendix B. ASCII and EBCDIC Character Sets

XL Fortran uses the ASCII character set as its collating sequence.

This table lists the standard ASCII characters in numerical order with the corresponding decimal and hexadecimal values. For convenience in working with programs that use EBCDIC character values, the corresponding information for EBCDIC characters is also included. The table indicates the control characters with “Ctrl-” notation. For example, the horizontal tab (HT) appears as “Ctrl-I”, which you enter by simultaneously pressing the Ctrl key and I key.

Table 19. Equivalent Characters in the ASCII and EBCDIC Character Sets

Decimal Value	Hex Value	Control Character	ASCII Symbol	Meaning	EBCDIC Symbol	Meaning
0	00	Ctrl-@	NUL	null	NUL	null
1	01	Ctrl-A	SOH	start of heading	SOH	start of heading
2	02	Ctrl-B	STX	start of text	STX	start of text
3	03	Ctrl-C	ETX	end of text	ETX	end of text
4	04	Ctrl-D	EOT	end of transmission	SEL	select
5	05	Ctrl-E	ENQ	enquiry	HT	horizontal tab
6	06	Ctrl-F	ACK	acknowledge	RNL	required new-line
7	07	Ctrl-G	BEL	bell	DEL	delete
8	08	Ctrl-H	BS	backspace	GE	graphic escape
9	09	Ctrl-I	HT	horizontal tab	SPS	superscript
10	0A	Ctrl-J	LF	line feed	RPT	repeat
11	0B	Ctrl-K	VT	vertical tab	VT	vertical tab
12	0C	Ctrl-L	FF	form feed	FF	form feed
13	0D	Ctrl-M	CR	carriage return	CR	carriage return
14	0E	Ctrl-N	SO	shift out	SO	shift out
15	0F	Ctrl-O	SI	shift in	SI	shift in
16	10	Ctrl-P	DLE	data link escape	DLE	data link escape
17	11	Ctrl-Q	DC1	device control 1	DC1	device control 1
18	12	Ctrl-R	DC2	device control 2	DC2	device control 2
19	13	Ctrl-S	DC3	device control 3	DC3	device control 3

© Copyright IBM Corp. 1990, 2000

Table 19. Equivalent Characters in the ASCII and EBCDIC Character Sets (continued)

Decimal Value	Hex Value	Control Character	ASCII Symbol	Meaning	EBCDIC Symbol	Meaning
20	14	Ctrl-T	DC4	device control 4	RES/ENP	restore/enable presentation
21	15	Ctrl-U	NAK	negative acknowledge	NL	new-line
22	16	Ctrl-V	SYN	synchronous idle	BS	backspace
23	17	Ctrl-W	ETB	end of transmission block	POC	program-operator communications
24	18	Ctrl-X	CAN	cancel	CAN	cancel
25	19	Ctrl-Y	EM	end of medium	EM	end of medium
26	1A	Ctrl-Z	SUB	substitute	UBS	unit backspace
27	1B	Ctrl-[	ESC	escape	CU1	customer use 1
28	1C	Ctrl-\	FS	file separator	IFS	interchange file separator
29	1D	Ctrl-]	GS	group separator	IGS	interchange group separator
30	1E	Ctrl-∧	RS	record separator	IRS	interchange record separator
31	1F	Ctrl-_	US	unit separator	IUS/ITB	interchange unit separator / intermediate transmission block
32	20		SP	space	DS	digit select
33	21		!	exclamation mark	SOS	start of significance
34	22		″	straight double quotation mark	FS	field separator
35	23		#	number sign	WUS	word underscore
36	24		$	dollar sign	BYP/INP	bypass/inhibit presentation
37	25		%	percent sign	LF	line feed
38	26		&	ampersand	ETB	end of transmission block
39	27		′	apostrophe	ESC	escape
40	28		(	left parenthesis	SA	set attribute
41	29		)	right parenthesis		
42	2A		*	asterisk	SM/SW	set model switch

Table 19. Equivalent Characters in the ASCII and EBCDIC Character Sets (continued)

Decimal Value	Hex Value	Control Character	ASCII Symbol	Meaning	EBCDIC Symbol	Meaning
43	2B		+	addition sign	CSP	control sequence prefix
44	2C		,	comma	MFA	modify field attribute
45	2D		-	subtraction sign	ENQ	enquiry
46	2E		.	period	ACK	acknowledge
47	2F		/	right slash	BEL	bell
48	30		0			
49	31		1			
50	32		2		SYN	synchronous idle
51	33		3		IR	index return
52	34		4		PP	presentation position
53	35		5		TRN	
54	36		6		NBS	numeric backspace
55	37		7		EOT	end of transmission
56	38		8		SBS	subscript
57	39		9		IT	indent tab
58	3A		:	colon	RFF	required form feed
59	3B		;	semicolon	CU3	customer use 3
60	3C		<	less than	DC4	device control 4
61	3D		=	equal	NAK	negative acknowledge
62	3E		>	greater than		
63						
3F		?	question mark	SUB	substitute	
64	40		@	at symbol	SP	space
65	41		A			
66	42		B			
67	43		C			
68	44		D			
69	45		E			
70	46		F			

Table 19. Equivalent Characters in the ASCII and EBCDIC Character Sets (continued)

Decimal Value	Hex Value	Control Character	ASCII Symbol	Meaning	EBCDIC Symbol	Meaning
71	47		G			
72	48		H			
73	49		I			
74	4A		J		¢	cent
75	4B		K		.	period
76	4C		L		<	less than
77	4D		M		(	left parenthesis
78	4E		N		+	addition sign
79	4F		O		|	logical or
80	50		P		&	ampersand
81	51		Q			
82	52		R			
83	53		S			
84	54		T			
85	55		U			
86	56		V			
87	57		W			
88	58		X			
89	59		Y			
90	5A		Z		!	exclamation mark
91	5B		[	left bracket	$	dollar sign
92	5C		\	left slash	*	asterisk
93	5D		]	right bracket	)	right parenthesis
94	5E		^	hat, circumflex	;	semicolon
95	5F		_	underscore	¬	logical not
96	60		`	grave	-	subtraction sign
97	61		a		/	right slash
98	62		b			
99	63		c			
100	64		d			
101	65		e			
102	66		f			

Table 19. Equivalent Characters in the ASCII and EBCDIC Character Sets (continued)

Decimal Value	Hex Value	Control Character	ASCII Symbol	Meaning	EBCDIC Symbol	Meaning
103	67		g			
104	68		h			
105	69		i			
106	6A		j		¦	split vertical bar
107	6B		k		,	comma
108	6C		l		%	percent sign
109	6D		m		_	underscore
110	6E		n		>	greater than
111	6F		o		?	question mark
112	70		p			
113	71		q			
114	72		r			
115	73		s			
116	74		t			
117	75		u			
118	76		v			
119	77		w			
120	78		x			
121	79		y		`	grave
122	7A		z		:	colon
123	7B		{	left brace	#	numbersign
124	7C		|	logical or	@	at symbol
125	7D		}	right brace	’	apostrophe
126	7E		˜	similar, tilde	=	equal
127	7F		DEL	delete	″	straight double quotation mark
128	80					
129	81				a	
130	82				b	
131	83				c	
132	84				d	
133	85				e	

Table 19. Equivalent Characters in the ASCII and EBCDIC Character Sets (continued)

Decimal Value	Hex Value	Control Character	ASCII Symbol	Meaning	EBCDIC Symbol	Meaning
134	86				f	
135	87				g	
136	88				h	
137	89				i	
138	8A					
139	8B					
140	8C					
141	8D					
142	8E					
143	8F					
144	90					
145	91				j	
146	92				k	
147	93				l	
148	94				m	
149	95				n	
150	96				o	
151	97				p	
152	98				q	
153	99				r	
154	9A					
155	9B					
156	9C					
157	9D					
158	9E					
159	9F					
160	A0					
161	A1				˜	similar, tilde
162	A2				s	
163	A3				t	
164	A4				u	
165	A5				v	

Table 19. Equivalent Characters in the ASCII and EBCDIC Character Sets (continued)

Decimal Value	Hex Value	Control Character	ASCII Symbol	Meaning	EBCDIC Symbol	Meaning
166	A6				w	
167	A7				x	
168	A8				y	
169	A9				z	
170	AA					
171	AB					
172	AC					
173	AD					
174	AE					
175	AF					
176	B0					
177	B1					
178	B2					
179	B3					
180	B4					
181	B5					
182	B6					
183	B7					
184	B8					
185	B9					
186	BA					
187	BB					
188	BC					
189	BD					
190	BE					
191	BF					
192	C0				{	left brace
193	C1				A	
194	C2				B	
195	C3				C	
196	C4				D	
197	C5				E	

Table 19. Equivalent Characters in the ASCII and EBCDIC Character Sets (continued)

Decimal Value	Hex Value	Control Character	ASCII Symbol	Meaning	EBCDIC Symbol	Meaning
198	C6				F	
199	C7				G	
200	C8				H	
201	C9				I	
202	CA					
203	CB					
204	CC					
205	CD					
206	CE					
207	CF					
208	D0				}	right brace
209	D1				J	
210	D2				K	
211	D3				L	
212	D4				M	
213	D5				N	
214	D6				O	
215	D7				P	
216	D8				Q	
217	D9				R	
218	DA					
219	DB					
220	DC					
221	DD					
222	DE					
223	DF					
224	E0				\	left slash
225	E1					
226	E2				S	
227	E3				T	
228	E4				U	
229	E5				V	

Table 19. Equivalent Characters in the ASCII and EBCDIC Character Sets (continued)

Decimal Value	Hex Value	Control Character	ASCII Symbol	Meaning	EBCDIC Symbol	Meaning
230	E6				W	
231	E7				X	
232	E8				Y	
233	E9				Z	
234	EA					
235	EB					
236	EC					
237	ED					
238	EE					
239	EF					
240	F0				0	
241	F1				1	
242	F2				2	
243	F3				3	
244	F4				4	
245	F5				5	
246	F6				6	
247	F7				7	
248	F8				8	
249	F9				9	
250	FA				|	vertical line
251	FB					
252	FC					
253	FD					
254	FE					
255	FF				EO	eight ones

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504–1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2–31 Roppongi 3–chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY

© Copyright IBM Corp. 1990, 2000

OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Lab Director
IBM Canada Limited
1150 Eglinton Avenue East
Toronto, Ontario
Canada
M3C 1H7

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

This software and documentation are based in part on the Fourth Berkeley Software Distribution under license from the Regents of the University of California. We acknowledge the following institution for its role in this product's development: the Electrical Engineering and Computer Sciences Department at the Berkeley campus.

Portions of this document are copied without fee from *High Performance Fortran Language Specification*, Version 1.1, CRPC-TR92225. Center for Research on Parallel Computation, Rice University. Houston, 1994. Permission is granted by Rice University.

OpenMP is a trademark of the OpenMP Architecture Review Board. Portions of this book may have been derived from the *OpenMP Fortran Language Application Program Interface*, Version 1.0 (Oct 1997) specification. Copyright 1997-98 OpenMP Architecture Review Board.

Trademarks and Service Marks

The following terms, used in this publication, are trademarks or service marks of the International Business Machines Corporation in the United States or other countries or both:

AIX	IBMLink	RS/6000
IBM	PROFS	RISC System 6000
RS/6000 SP	POWER Parallel	PowerPC
POWER2		

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

Glossary

This glossary defines terms that are commonly used in this book. It includes definitions developed by the American National Standards Institute (ANSI) and entries from the *IBM Dictionary of Computing*

A

actual argument. An expression, variable, procedure, or alternate return specifier that is specified in a procedure reference.

alphabetic character. A letter or other symbol, excluding digits, used in a language. Usually the uppercase and lowercase letters A through Z plus other special symbols (such as _) allowed by a particular language.

alphanumeric. Pertaining to a character set that contains letters, digits, and usually other characters, such as punctuation marks and mathematical symbols.

American National Standard Code for Information Interchange (ASCII). The code developed by ANSI for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII character set consists of 7-bit control characters and symbolic characters.

American National Standards Institute (ANSI). An organization sponsored by the Computer and Business Equipment Manufacturers Association through which accredited organizations create and maintain voluntary industry standards.

ANSI. American National Standards Institute.

argument. An actual argument or a dummy argument.

argument association. The relationship between an actual argument and a dummy argument during the execution of a procedure reference.

arithmetic constant. A constant of type integer, real, or complex.

arithmetic expression. One or more arithmetic operators and arithmetic primaries, the evaluation of which produces a numeric value. An arithmetic expression can be an unsigned arithmetic constant, the name of an arithmetic constant, or a reference to an arithmetic variable, function reference, or a combination of such primaries formed by using arithmetic operators and parentheses.

arithmetic operator. A symbol that directs the performance of an arithmetic operation. The intrinsic arithmetic operators are:

```
+      addition
-      subtraction
*      multiplication
/      division
**     exponentiation
```

array. An entity that contains an ordered group of scalar data. All objects in an array have the same data type and type parameters.

array declarator. The part of a statement that describes an array used in a program unit. It indicates the name of the array, the number of dimensions it contains, and the size of each dimension.

array element. A single data item in an array, identified by the array name followed by one or more integer expressions called subscript expressions that indicate its position in the array.

array name. The name of an ordered set of data items.

array pointer. A pointer to an array.

array section. A subobject that is an array and is not a structure component.

ASCII. American National Standard Code for Information Interchange.

© Copyright IBM Corp. 1990, 2000

assignment statement. An assignment statement can be intrinsic or defined. An intrinsic assignment stores the value of the right operand in the storage location of the left operand.

attribute. A property of a data object that may be specified in a type declaration statement, attribute specification statement, or through a default setting.

automatic parallelization. The process by which the compiler attempts to parallelize both explicitly coded **DO** loops, as well as those generated by the compiler for array language.

B

binary constant. A constant that is made of one or more binary digits (0 and 1).

bind. To relate an identifier to another object in a program; for example, to relate an identifier to a value, an address or another identifier, or to associate formal parameters and actual parameters.

blank common. An unnamed common block.

block data subprogram. A subprogram headed by a **BLOCK DATA** statement and used to initialize variables in named common blocks.

byte constant. A named constant that is of type byte.

byte type. A data type representing a one-byte storage area that can be used wherever a **LOGICAL(1)**, **CHARACTER(1)**, or **INTEGER(1)** can be used.

C

character constant. A string of one or more alphabetic characters enclosed in apostrophes or double quotation marks.

character expression. A character object, a character-valued function reference, or a sequence of them separated by the concatenation operator, with optional parentheses.

character operator. A symbol that represents an operation, such as concatenation (//) to be performed on character data.

character set. All the valid characters for a programming language or for a computer system.

character string. A sequence of consecutive characters.

character substring. A contiguous portion of a character string.

character type. A data type that consists of alphanumeric characters. See also *data type*.

chunk. A subset of consecutive loop iterations.

collating sequence. The sequence in which characters are ordered within the computer for sorting, combining, or comparing. The AIX collating sequence used by XL Fortran is ASCII.

comment. A language construct for the inclusion of text in a program that has no effect on the execution of the program.

common block. A storage area that may be referred to by a calling program and one or more subprograms.

compiler directive. Source code that controls what XL Fortran does rather than what the user program does.

complex constant. An ordered pair of real or integer constants separated by a comma and enclosed in parentheses. The first constant of the pair is the real part of the complex number; the second is the imaginary part.

complex number. A number consisting of an ordered pair of real numbers, expressible in the form **a+bi**, where **a** and **b** are real numbers and **i** squared equals -1.

complex type. A data type that represents the values of complex numbers. The value is expressed as an ordered pair of real data items separated by a comma and enclosed in parentheses. The first item represents the real

part of the complex number; the second represents the imaginary part.

conformance. An executable program conforms to the Fortran 90 Standard if it uses only those forms and relationships described therein and if the executable program has an interpretation according to the Fortran 90 Standard. A program unit conforms to the Fortran 90 Standard if it can be included in an executable program in a manner that allows the executable program to be standard-conforming. A processor conforms to the standard if it executes standard-conforming programs in a manner that fulfills the interpretations prescribed in the standard.

connected unit. In XL Fortran, a unit that is connected to a file in one of three ways: explicitly via the **OPEN** statement to a named file, implicitly, or by preconnection.

constant. A data object with a value that does not change. Contrast with *variable*. The four classes of constants specify numbers (arithmetic), truth values (logical), character data (character), and typeless data (hexadecimal, octal, and binary).

construct. A sequence of statements starting with a **SELECT CASE**, **DO**, **IF**, or **WHERE** statement and ending with the corresponding terminal statement.

continuation line. Continues a statement beyond its initial line.

control statement. A statement that is used to alter the continuous sequential invocation of statements; a control statement may be a conditional statement, such as **IF**, or an imperative statement, such as **STOP**.

D

data object. A variable, constant, or subobject of a constant.

data transfer statement. A **READ**, **WRITE**, or **PRINT** statement.

data type. The properties and internal representation that characterize data and functions. The intrinsic types are integer, real, complex, logical, and character.

debug line. Allowed only for fixed source form, a line containing source code taht is to be used for debugging. Debug lines are defined by a `D` in column 1. The handling of debug lines is controlled by the **-qdlines** compiler option.

definable. A variable is definable if its value can be changed by the appearance of its name or designator on the left of an assignment statement.

delimiters. A pair of parentheses or slashes (or both) used to enclose syntactic lists.

derived type. A type whose data have components, each of which is either of intrinsic type or of another derived type.

digit. A character that represents a nonnegative integer. For example, any of the numerals from 0 through 9.

directive. A type of comment that provides instructions and information to the compiler.

DO loop. A range of statements invoked repetitively by a **DO** statement.

DO variable. A variable, specified in a **DO** statement, that is initialized or incremented prior to each occurrence of the statement or statements within a **DO** range. It is used to control the number of times the statements within the range are executed.

DOUBLE PRECISION constant. A constant of type real with twice the precision of the default real precision.

dummy argument. An entity whose name appears in the parenthesized list following the procedure name in a **FUNCTION**, **SUBROUTINE**, **ENTRY**, or statement function statement.

dynamic extent. The dynamic extent of a directive includes the lexical extent of the directive and all subprograms called from within the lexical extent.

edit descriptors. In Fortran, abbreviated keywords that control the formatting of integer, real, and complex data.

E

elemental. An adjective applied to an intrinsic operation, procedure or assignment that is applied independently to elements of an array or corresponding elements of a set of conformable arrays and scalars.

embedded blanks. Blanks that are surrounded by any other characters.

entity. A general term for the following: a program unit, procedure, operator, interface block, common block, external unit, statement function, type, named variable, expression, component of a structure, named constant, statement label, construct, or namelist group.

executable program. A program that can be executed as a self-contained procedure. It consists of a main program and, optionally, modules, subprograms and non-Fortran external procedures.

executable statement. A statement that causes an action to be taken by the program; for example, to calculate, test conditions, or alter normal sequential execution.

existing unit. A valid unit number that is system-specific.

explicit interface. For a procedure referenced in a scoping unit, the property of being an internal procedure, module procedure, intrinsic procedure, external procedure that has an interface block, recursive procedure reference in its own scoping unit, or dummy procedure that has an interface block.

expression. A sequence of operands, operators, and parentheses. It may be a variable, constant, function reference, or it may represent a computation.

extended-precision constant. A processor approximation to the value of a real number that occupies 16 consecutive bytes of storage.

external procedure. A procedure that is defined by an external subprogram or by a means other than Fortran.

F

field. An area in a record used to contain a particular category of data.

file. A sequence of records. If the file is located in internal storage, it is an internal file; if it is on an input/output device, it is an external file.

floating-point number. A real number represented by a pair of distinct numerals. The real number is the product of the fractional part, one of the numerals, and a value obtained by raising the implicit floating-point base to a power indicated by the second numeral.

format. (1) A defined arrangement of such things as characters, fields, and lines, usually used for displays, printouts, or files. (2) To arrange such things as characters, fields, and lines.

formatted data. Data that is transferred between main storage and an input/output device according to a specified format. See also *list-directed data* and *unformatted data*.

FORmula TRANslation (Fortran). A high-level programming language used primarily for scientific, engineering, and mathematical applications.

Fortran. FORmula TRANslation.

function. A procedure that returns the value of a single variable and that usually has a single exit. See also *function subprogram*, *intrinsic function*, and *statement function*.

G

generic identifier. A lexical token that appears in an **INTERFACE** statement and is associated with all the procedures in an interface block.

H

hexadecimal. Pertaining to a system of numbers to the base sixteen; hexadecimal digits range from 0 (zero) through 9 (nine) and A (ten) through F (fifteen).

hexadecimal constant. A constant, usually starting with special characters, that contains only hexadecimal digits.

Hollerith constant. A string of any characters capable of representation by XL Fortran and preceded with **nH**, where n is the number of characters in the string.

host. A main program or subprogram that contains an internal procedure is called the host of the internal procedure. A module that contains a module procedure is called the host of the module procedure.

host association. The process by which an internal subprogram, module subprogram, or derived-type definition accesses the entities of its host.

I

implicit interface. A procedure referenced in a scoping unit other than its own is said to have an implicit interface if the procedure is an external procedure that does not have an interface block, a dummy procedure that does not have an interface block, or a statement function.

implied DO. An indexing specification (similar to a **DO** statement, but without specifying the word **DO**) with a list of data elements, rather than a set of statements, as its range.

input/output (I/O). Pertaining to either input or output, or both.

input/output list. A list of variables in an input or output statement specifying the data to be read or written. An output list can also contain a constant, an expression involving operators or function references, or an expression enclosed in parentheses.

integer constant. An optionally signed digit string that contains no decimal point.

interface block. A sequence of statements from an **INTERFACE** statement to the corresponding **END INTERFACE** statement.

interface body. A sequence of statements in an interface block from a **FUNCTION** or **SUBROUTINE** statement to the corresponding **END** statement.

interference. When two iterations within a **DO** loop have dependencies upon one another. For more information, see “ASSERT” on page 445.

intrinsic. An adjective applied to types, operations, assignment statements, and procedures that are defined by Fortran 90 and can be used in any scoping unit without further definition or specification.

I/O. Input/output.

K

keyword. (1) A statement keyword is a word that is part of the syntax of a statement (or directive) and that may be used to identify the statement. (2) An argument keyword specifies a name for a dummy argument.

kind type parameter. A parameter whose values label the available kinds of an intrinsic type.

L

lexical extent. The lexical extent of a directive includes all code that appears directly within the directive construct.

lexical token. A sequence of characters with an indivisible interpretation.

list-directed. A predefined input/output format that depends on the type, type parameters, and values of the entities in the data list.

literal. A symbol or a quantity in a source program that is itself data, rather than a reference to data.

literal constant. In Fortran, a lexical token that directly represents a scalar value of intrinsic type.

logical constant. A constant with a value of either `true` or `false` (or T or F).

logical operator. A symbol that represents an operation on logical expressions:

```
.NOT.    (logical negation)
.AND.    (logical conjunction)
.OR.     (logical union)
.EQV.    (logical equivalence)
.NEQV.   (logical nonequivalence)
.XOR.    (logical exclusive disjunction)
```

loop. A statement block that executes repeatedly.

M

main program. The first program unit to receive control when a program is run. Contrast with *subprogram.*

master thread. The head process of a group of threads.

module. A program unit that contains or accesses definitions to be accessed by other program units.

mutex. The word mutex is shorthand for a primitive object that provides MUTual EXclusion between threads. A mutual exclusion (mutex) is used cooperatively between threads to ensure that only one of the cooperating threads is allowed to access the data or run certain application code at a time.

N

name. A lexical token consisting of a letter followed by up to 249 alphanumeric characters (letters, digits, and underscores). Note that in FORTRAN 77, this was called a symbolic name.

named common. A separate, named common block consisting of variables.

namelist group name. The first parameter in the NAMELIST statement that names a list of names to be used in READ, WRITE, and PRINT statements.

nest. To incorporate a structure or structures of some kind into a structure of the same kind. For example, to nest one loop (the nested loop) within another loop (the nesting loop); to nest one subroutine (the nested subroutine) within another subroutine (the nesting subroutine).

nonexecutable statement. A statement that describes the characteristics of a program unit, data, editing information, or statement functions, but does not cause any action to be taken by the program.

nonexisting file. A file that does not physically exist on any accessible storage medium.

numeric constant. A constant that expresses an integer, real, complex, or byte number.

O

octal. Pertaining to a system of numbers to the base eight; the octal digits range from 0 (zero) through 7 (seven).

octal constant. A constant that is made of octal digits.

operator. In Fortran, a specification of a particular computation involving one or two operands.

P

pad. To fill unused positions in a field or character string with dummy data, usually zeros or blanks.

pointer. A variable that has the **POINTER** attribute. A pointer must not be referenced or defined unless it is pointer associated with a target. If it is an array, it does not have a shape unless it is pointer-associated.

preconnected file. A file that is connected to a unit at the beginning of execution of the executable program. Standard error, standard input, and standard output are preconnected files (units 0, 5 and 6, respectively).

predefined convention. The implied type and length specification of a data object, based on the initial character of its name when no explicit specification is given. The initial characters I through N imply type integer of length 4; the initial characters A through H, O through Z, $, and _ imply type real of length 4.

present. A dummy argument is present in an instance of a subprogram if it is associated with an actual argument and the actual argument is a dummy argument that is present in the invoking procedure or is not a dummy argument of the invoking procedure.

primary. The simplest form of an expression: an object, array constructor, structure constructor, function reference, or expression enclosed in parentheses.

procedure. A computation that may be invoked during program execution. It may be a function or subroutine. It may be an intrinsic procedure, external procedure, module procedure, internal procedure, dummy procedure, or statement function. A subprogram may define more than one procedure if it contains **ENTRY** statements.

program unit. A main program or subprogram.

pure. An attribute of a procedure that indicates there are no side effects.

R

random access. An access method in which records can be read from, written to, or removed from a file in any order.

rank. In Fortran, the number of dimensions of an array.

real constant. A string of decimal digits that expresses a real number. A real constant must contain a decimal point, a decimal exponent, or both.

record. A sequence of values that is treated as a whole within a file.

relational expression. An expression that consists of an arithmetic or character expression, followed by a relational operator, followed by another arithmetic or character expression.

relational operator. The words or symbols used to express a relational condition or a relational expression:

```
.GT.       greater than
.GE.       greater than or equal to
.LE.       less than or equal to
.EQ.       equal to
.NE.       not equal to
```

result variable. The variable that returns the value of a function.

return specifier. An argument sepcified for a statement, such as CALL, that indicates to which statement label control should return, depending on the action specified by the subroutine in the RETURN statement.

S

scalar. (1) A single datum that is not an array. (2) Not having the property of being an array.

scale factor. A number indicating the location of the decimal point in a real number (and, on input, if there is no exponent, the magnitude of the number).

scope. That part of an executable program within which a lexical token has a single interpretation.

scope attribute. That part of an executable program within which a lexical token has a single interpretation of a particular named property or entity.

scoping unit. (1) A derived-type definition. (2) An interface body, excluding any derived-type definitions and interface bodies contained within it. (3) A program unit or subprogram, excluding derived-type definitions, interface bodies, and subprograms contained within it.

selector. A pointer, pointing device, or selection cursor.

sequential access. An access method in which records are read from, written to, or removed from a file based on the logical order of the records in the file.

SMP. Symmetric Multiprocessing

specification statement. One of the set of statements that provides information about the data used in the source program. The statement could also supply information to allocate data storage.

statement. A language construct that represents a step in a sequence of actions or a set of declarations. Statements fall into two broad classes: executable and nonexecutable.

statement function. A name, followed by a list of dummy arguments, that is equated with an intrinsic or derived-type expression, and that can be used as a substitute for the expression throughout the program.

statement label. A number from one through five digits that is used to identify a statement. Statement labels can be used to transfer control, to define the range of a **DO**, or to refer to a **FORMAT** statement.

storage association. The relationship between two storage sequences if a storage unit of one is the same as the storage unit of the other.

structure. A scalar data object of derived type.

structure component. The part of a data object of derived-type corresponding to a component of its type.

subobject. A portion of a named data object that may be referenced or defined independently of other portions. It can be an array element, array section, structure component, or substring.

subprogram. A function subprogram or a subroutine subprogram. Note that in FORTRAN 77, a block data program unit was called a subprogram.

subroutine. A procedure that is invoked by a **CALL** statement or defined assignment statement.

subscript. A subscript quantity or set of subscript quantities enclosed in parentheses and used with an array name to identify a particular array element.

substring. A contiguous portion of a scalar character string. (Although an array section can specify a substring selector, the result is not a substring.)

Symmetric Multiprocessing (SMP). Any processor can be substituted for any other processor without affecting the architecture of the machine.

T

target. A named data object specified to have the **TARGET** attribute, a data object created by an **ALLOCATE** statement for a pointer, or a subobject of such an object.

thread. A collection of processes whose order determines the process eligible for execution. A thread is the element that is scheduled, and to which resources such as time slices, locks and queues may be assigned.

thread visible variable. A variable that is visible to more than one thread. See “FLUSH” on page 462 for more information.

time slice. An interval of time on the processing unit allocated for use in performing a task. After the interval has expired, processing unit time is allocated to another task, so a task cannot monopolize processing unit time beyond a fixed limit.

token. Ina programming language, a character string, in a particular format, that has some defined significance.

type declaration statement. Specifies the type, length, and attributes of objects and functions. Objects can be assigned initial values.

U

unformatted record. A record that is transmitted unchanged between internal and external storage.

unit. A means of referring to a file to use in input/output statements. A unit can be connected or not connected to a file. If connected, it refers to a file. The connection is symmetric: that is, if a unit is connected to a file, the file is connected to the unit.

use association. The association of names in different scoping units specified by a **USE** statement.

V

variable. A data object whose value can be defined and redefined during the execution of an executable program. It may be a named data object, array element, array section, structure component, or substring. Note that in FORTRAN 77, a variable was always scalar and named.

Z

zero-length character. A character object that has a length of 0 and is always defined.

zero-sized array. An array that has a lower bound that is greater than its corresponding upper bound. The array is always defined.

INDEX

Special Characters

+, -, *, /, ** arithmetic operators 91
// (concatenation) operator 94
:: (double colon) separator 254
"" (double quotation mark) editing 221
' (apostrophe) editing 221
: (colon) editing 221
$ (dollar) editing 221
/ (slash) editing 220
* comment lines 17
! inline comments 16, 17
@PROCESS compiler directive 490
; statement separator 18, 20
%VAL and %REF functions 165
#line compiler directive 470
_OPENMP C preprocessor macro 24

A

A (character) editing 205
ABORT intrinsic subroutine 524
ABS
  initializing expressions 88
  intrinsic function 524
  specific name 525
access, inquiring about 332
ACCESS specifier
  of INQUIRE statement 332
  of OPEN statement 355
accessibility
  private 370
  public 373
ACHAR intrinsic function 525
ACOS
  intrinsic function 525
  specific name 526
ACOSD
  intrinsic function 526
  specific name 527
ACTION specifier
  of INQUIRE statement 332
  of OPEN statement 355
actual arguments
  definition of 751
  specification 161
  specifying procedure names as 310
addition arithmetic operator 91
ADJUSTL intrinsic function 527
ADJUSTR intrinsic function 527
ADVANCE specifier
  of READ statement 375
  of WRITE statement 416
AFFINITY scheduling type of SCHEDULE directive 491
AIMAG
  initializing expressions 88
  intrinsic function 527
  specific name 528
AINT
  intrinsic function 528
  specific name 529
AIX Pthreads Library 677
alarm_ service and utility subprogram 644
ALGAMA specific name 578
ALL array intrinsic function 529
ALLOCATABLE attribute 245
ALLOCATE statement 246
ALLOCATED array intrinsic function 248, 529
allocation status 58
ALOG specific name 582
ALOG10 specific name 583
alphabetic character, definition of 751
alphanumeric, definition of 751
alternate entry point 303
alternate return
  point 163
  specifier 161, 172
AMAX0 specific name 587
AMAX1 specific name 587
AMIN0 specific name 592
AMIN1 specific name 592
AMOD specific name 596
AND logical operator 97
AND specific name 567
ANINT intrinsic function 530
ANINT specific name 531
ANSI, definition of 751
ANY array intrinsic function 531
apostrophe (') editing 221
arguments
  definition of 751
  keywords 162
  specification 161
arithmetic
  expressions 91
  operators 91
  relational expressions 94
  type
    complex 33
    integer 29
    real 31
arithmetic IF statement 327
arrays
  adjustable 67
  allocatable 70
  array pointers 71
  assumed-shape 68
  assumed-size 71, 89
  automatic 66
  bounds 63
  constructors 81
  declarators 65
  decription 63
  deferred-shape 69
  elements 73
  explicit-shape 66
  extents 64
  pointee 67
  pointer 71
  rank 64
  sections 74
  shape 64
  size 64
  specification of 65
  zero-sized 63
asa command 183
ASCII
  character set 13, 737
  definition of 751
ASIN
  intrinsic function 532
  specific name 532
ASIND
  intrinsic function 532
  specific name 533
ASSERT compiler directive 445
ASSIGN statement 248
assigned GO TO statement 324
assignment
  defined 151
  intrinsic 103
  masked array 108

© Copyright IBM Corp. 1990, 2000

assignment (continued)
  pointer 117
  statements
    described 103
    statement label (ASSIGN) 248
ASSOCIATED intrinsic function 248, 533
association
  argument 163
  common 268
  description 137
  entry 322
  equivalence 307
  host 137
  integer pointer 140
  pointer 139
  use 139
asterisk as dummy argument 163, 172
ASYNCH specifier
  of INQUIRE statement 332
  of OPEN statement 355
asynchronous I/O
  data transfer and 189
  INQUIRE statement and 333
  OPEN statement and 355
  READ statement and 377
  WAIT statement and 413
  WRITE statement and 418
ATAN
  intrinsic function 534
  specific name 534
ATAN2
  intrinsic function 535
  specific name 536
ATAN2D
  intrinsic function 536
  specific name 537
ATAND
  intrinsic function 534
  specific name 535
ATOMIC compiler directive 448
attributes
  ALLOCATABLE 245
  AUTOMATIC 250
  description 244
  DIMENSION 283
  EXTERNAL 310
  INTENT 342
  INTRINSIC 345
  OPTIONAL 361
  PARAMETER 362
  POINTER 364
  PRIVATE 370
attributes (continued)
  PUBLIC 373
  SAVE 388
  STATIC 394
  TARGET 398
  VOLATILE 410
AUTOMATIC attribute 250
automatic object 29

B

B (binary) editing 206
BACKSPACE statement 251
BARRIER compiler directive 450
bic_ service and utility subprogram 644
binary
  constants 48
  editing (B) 206
  operations 85
bis_ service and utility subprogram 645
bit_ service and utility subprogram 645
BIT_SIZE
  intrinsic, constant expressions and 87
  intrinsic function 88, 537
blank
  common block 268
  editing 222
  interpretation during formatting, setting 222
  null (BN) editing 222
  specifier
    of INQUIRE statement (BLANK) 332
    of OPEN statement (BLANK) 355
  zero (BZ) editing 222
block
  cyclic scheduling 493
  ELSE 122
  ELSE IF 122
  IF 122, 328
  statement 121
block data
  program unit 156
  statement (BLOCK DATA) 252
BN (blank null) editing 222
branching control 131
BTEST
  intrinsic function 537
  specific name 538
byte named constants 103
BYTE type declaration statement 253
BZ (blank zero) editing 222

C

CABS specific name 525
CALL statement 257
CASE
  construct 123, 258
  statement 258
CCOS specific name 541
CDABS specific name 525
CDCOS specific name 541
CDEXP specific name 557
CDLOG specific name 582
CDSIN specific name 621
CDSQRT specific name 625
CEILING intrinsic function 538
CEXP specific name 557
CHAR
  intrinsic function 539
  specific name 539
character
  editing
    (A) 205
    (Q), count 217
  expressions 93
  format specification 319
  multibyte 37
  operator 94
  relational expressions 95
  set 13
  string edit descriptor 203, 319
  substrings 37
character-string editing 221
CHARACTER type declaration statement 260
chtz command 547
chunk
  definition of 752
  SCHEDULE directive and 491
clock_ service and utility subprogram 645
CLOG specific name 582
CLOSE statement 265
clr_fpscr_flags service and utility subprogram 645
CMPLX
  initializing expressions 88
  intrinsic function 539
  specific name 540
CNCALL compiler directive 452
CNVERR run-time option
  conversion errors and 198
  implied-DO list and 380, 420

collating sequence 13
colon (:) editing 221
comment lines
  description 16
  fixed source form format 17
  free source form input format 19
  order within a program unit 25
common
  association 268
  block 14, 267
common blocks
  in data scope attribute clauses 430
COMMON statement 267
communication between program units
  using arguments 161
  using common blocks 267
  using modules 153
compatibility across standards 731
compiler directives 423
compiler options
  -I 465
  –qcclines 23
  -qalias 165
  -qautodbl 523
  -qci 464
  -qctyplss
    and the CASE statement 260
    character constants and 37, 102
    typeless constants and 49
  -qddim 67, 367
  -qdirective 506
  -qdlines 18
  -qescape
    and Hollerith constants 48
    apostrophe editing and 221
    double quotation mark editing and 221
    H editing and 223
  -qextname 642
  -qfixed 17
  -qintlog 101, 148
  -qintsize
    integer default size and 29, 35
    intrinsic procedure return types and 523
  -qlog4 101
  -qmbcs 221, 223
  -qmixed 14, 465
  -qnoescape 37
  -qnosave 60, 331
compiler options (continued)
  -qnullterm 37
  -qposition 191, 355
  -qqcount 217
  -qrealsize 30, 523
  -qrecur 175
    CALL statement and 257
    ENTRY statement and 306
    FUNCTION statement and 323
    SUBROUTINE statement and 397
  -qsave 60, 331
  -qsigtrap 621
  -qundef 330
  -qxflag=oldtab 17
  -qxlf77
    binary editing and 206, 211, 219
    hexadecimal editing and 219
    octal editing and 216
    OPEN statement and 359
    real and complex editing and 213
  -qxlf90 204, 619
  -qzerosize 38
  -U 642
complex
  data type 34
  editing 205
COMPLEX type declaration statement 270
component designator 44
computed GO TO statement 325
concatenation operator 94
conditional
  INCLUDE 465
  vector merge intrinsic functions 546
conditional compilation 22
conformable arrays 82, 519
CONJG
  initializing expressions 88
  intrinsic function 540
  specific name 541
conjunction, logical 97
constants
  arithmetic
    complex 33
    integer 29
    real 31
  binary 48
  byte named 103
  character 35
  description 28
constants (continued)
  expressions 87
  hexadecimal 46
  Hollerith 48
  logical 35
  octal 47
  type parameters, specifiers and 28
  typeless 46
construct
  CASE 123
  control 8
  DO 126
  DO WHILE 130
  FORALL 113
  IF 121
  WHERE 106
construct entities 134, 136
construct name 136
constructors
  for arrays 81
  for complex objects 33
  for structures 44
CONTAINS statement 274
continuation
  character 17, 22
  lines 16
CONTINUE statement 275
control
  construct 8
  description 121
  edit descriptors 202, 318
  format 226
  statements
    arithmetic IF 327
    assigned GO TO 324
    block IF 328
    computed GO TO 325
    CONTINUE 275
    DO 284
    DO WHILE 286
    END 296
    logical IF 329
    PAUSE 364
    STOP 395
    unconditional GO TO 326
  transfer of 25
control mask 108
COPYIN clause 441
COS
  intrinsic function 541
  specific name 541
COSD
  intrinsic function 541
  specific name 542

COSH
  intrinsic function 542
  specific name 542
COUNT array intrinsic function 542
CPU_TIME intrinsic function 543
cpu_time_type run-time option 543
CQABS specific name 525
CQCOS specific name 541
CQEXP specific name 557
CQLOG specific name 582
CQSIN specific name 621
CQSQRT specific name 625
CRITICAL compiler directive 453
CSHIFT array intrinsic function 545
CSIN specific name 621
CSQRT specific name 625
ctime_ service and utility subprogram 646
CVMGM, CVMGN, CVMGP, CVMGT, CVMGZ intrinsic functions 546
CYCLE statement 276

D

D (double precision) editing 207
D debug lines 16
DABS specific name 525
DACOS specific name 526
DACOSD specific name 527
DASIN specific name 532
DASIND specific name 533
data
  edit descriptors 201, 205, 316
  objects 28
  statement (DATA) 277
  type
    derived 39
  types
    conversion rules 92
    description 27
    intrinsic 29
    predefined conventions 52
data scope attribute clauses
  COPYIN clause 441
  DEFAULT clause 442
  discussion 430
  FIRSTPRIVATE clause 437
  LASTPRIVATE clause 437
  PRIVATE clause 436
  REDUCTION clause 438
  SHARED clause 443
data transfer
  asynchronous 189
  executing 188
data transfer *(continued)*
  statement
    PRINT 368
    READ 374
    WRITE 416
DATAN specific name 534
DATAN2 specific name 536
DATAN2D specific name 537
DATAND specific name 535
DATE_AND_TIME intrinsic subroutine 547
date service and utility subprogram 646
DBLE
  initializing expressions 88
  intrinsic function 549
  specific name 550
DBLEQ specific name 550
DCMPLX
  initializing expressions 88
  intrinsic function 550
  specific name 550
DCONJG specific name 541
DCOS specific name 541
DCOSD specific name 542
DCOSH specific name 542
DDIM specific name 552
DEALLOCATE statement 280
debug lines 16, 18
declarators
  array 65
  scoping level 134
DEFAULT clause 442
default typing 52
deferred-shape arrays 69
defined assignment 151
defined operations 99
defined operators 150
definition status 53
DELIM specifier
  of INQUIRE statement 332
  of OPEN statement 355
DERF specific name 556
DERFC specific name 557
derived-type statement 282
derived types
  array structure components 79
  description 39
  determining the type of 42
  scalar structure components 43
  structure components 41
  structure constructor 44
designators
  for array elements 73
  for components 44
DEXP specific name 557
DFLOAT specific name 550
digit-string 27
digits 13
DIGITS intrinsic function 551
DIM
  initializing expressions 88
  intrinsic function 551
  specific name 552
DIMAG specific name 528
DIMENSION attribute 283
dimension bound expression 63
dimensions of an array 64
DINT specific name 529
DIRECT specifier, of INQUIRE statement 332
directive lines 16
directives
  @PROCESS 490
  #line 470
  ASSERT 445
  ATOMIC 448
  BARRIER 450
  CNCALL 452
  CRITICAL 453
  discussion 423
  DO (work-sharing) 455
  DO SERIAL 460
  EJECT 461
  END CRITICAL 453
  END DO 455
  END MASTER 472
  END ORDERED 473
  END PARALLEL 476
  END PARALLEL DO 479
  END PARALLEL SECTIONS 483
  END SECTIONS 498
  FLUSH 462
  INCLUDE 464
  INDEPENDENT 466
  MASTER 472
  ORDERED 473
  PARALLEL 476
  PARALLEL DO 479
  PARALLEL SECTIONS 483
  PERMUTATION 487
  PREFETCH_BY_LOAD 488
  PREFETCH_FOR_LOAD 488
  PREFETCH_FOR_STORE 488
  SCHEDULE 491
  SECTIONS 498
  SINGLE / END SINGLE 502
  SOURCEFORM 506
  THREADLOCAL 507

directives *(continued)*
THREADPRIVATE 510
UNROLL 514
disconnection, closing files and 187
disjunction, logical 96, 97
division arithmetic operator 91
DLGAMA specific name 578
DLOG specific name 582
DLOG10 specific name 583
DMAX1 specific name 587
DMIN1 specific name 592
DMOD specific name 596
DNINT specific name 531
DO
loop 126, 284
statement 126, 284
DO (work-sharing) compiler directive
discussion 455
SCHEDULE clause 455
DO compiler directive
ORDERED clause 456
DO SERIAL compiler directive 460
DO WHILE
construct 130
loop 286
statement 286
dollar ($) editing 221
DONE specifier, of WAIT statement 413
DOT_PRODUCT array intrinsic function 552
DOUBLE COMPLEX type declaration statement 287
double precision (D) editing 207
DOUBLE PRECISION type declaration statement 290
double quotation mark ("") editing 221
DPROD
initializing expressions 88
intrinsic function 553
specific name 553
DREAL specific name 612
DSIGN specific name 620
DSIN specific name 621
DSIND specific name 622
DSINH specific name 622
DSQRT specific name 625
DTAN specific name 630
DTAND specific name 631
DTANH specific name 631
dtime_ service and utility subprogram 646
dummy argument
asterisk as 172
definition of 753
description 162
intent attribute and 166
procedure as 171
variable as 168
dummy procedure 171
dynamic extent, definition of 754
DYNAMIC scheduling type of SCHEDULE directive 491

E

E (real with exponent) editing 207
EBCDIC character set 737
edit descriptors
character string 203, 319
control (nonrepeatable) 202, 318
data (repeatable) 201, 316
names and 14
numeric 204
editing
' (apostrophe) 221
: (colon) 221
$ (dollar) 221
" (double quotation mark) 221
/ (slash) 220
A (character) 205
B (binary) 206
BN (blank null) 222
BZ (blank zero) 222
character count Q 217
character-string 221
complex 205
D (double precision) 207
discussion 204
E (real with exponent) 207
EN 209
ES 210
F (real without exponent) 211
G (general) 212
H 223
I (integer) 214
L (logical) 215
O (octal) 216
P (scale factor) 224
Q (extended precision) 207
S, SS, and SP (sign control) 224
T, TL, TR, and X (positional) 225
Z (hexadecimal) 219
efficient floating-point control and inquiry procedures
clr_fpscr_flags 645
discussion 639
efficient floating-point control and inquiry procedures *(continued)*
fp_trap 650
get_fpscr 652
get_fpscr_flags 653
get_round_mode 654
set_fpscr 662
set_fpscr_flags 662
set_round_mode 663
EJECT compiler directive 461
ELEMENTAL 178
elemental intrinsic procedures 519
elemental procedures 178
ELSE
block 122
statement 122, 294
ELSE IF
block 122
statement 122, 294
ELSEWHERE statement 106, 295
EN editing 209
encapsulation 8
END CRITICAL compiler directive 453
END DO compiler directive 455
END DO statement 126, 298
END FORALL statement 298
END IF statement 122, 298
END INTERFACE statement 144, 300
END MASTER compiler directive 472
end-of-file conditions 193
end-of-record, preventing with $ editing 221
end-of-record conditions 193
END ORDERED compiler directive 473
END PARALLEL compiler directive 476
END PARALLEL DO compiler directive 479
END PARALLEL SECTIONS compiler directive 483
END SECTIONS compiler directive 498
END SELECT statement 298
END specifier
of READ statement 375
of WAIT statement 413
END statement 296
END TYPE statement 301
END WHERE statement 106, 298
endfile records 184
ENDFILE statement 302

entities, scope of 134
entry
  association 322
  name 304
  statement (ENTRY) 303
EOR specifier, of READ statement 375
EOSHIFT array intrinsic function 553
EPSILON intrinsic function 555
equivalence
  logical 97
EQUIVALENCE
  association 306
  restriction on COMMON and 269
EQUIVALENCE statement 306
EQV logical operator 97
ERF
  intrinsic function 556
  specific name 556
ERFC
  intrinsic function 556
  specific name 557
ERR_RECOVERY run-time option
  BACKSPACE statement and 252
  conversion errors and 198
  EDNFILE statement and 303
  Fortran 90 language errors and 199
  Fortran 95 language errors and 199
  OPEN statement and 360
  READ statement and 380
  REWIND statement and 388
  severe errors and 194
  WRITE statement and 420
ERR specifier
  of BACKSPACE statement 251
  of CLOSE statement 265
  of ENDFILE statement 302
  of INQUIRE statement 332
  of OPEN statement 355
  of READ statement 375
  of REWIND statement 387
  of WAIT statement 413
  of WRITE statement 416
error conditions 193
errors
  catastrophic 193
  conversion 198
  Fortran 90 language 199
  Fortran 95 language 199
  recoverable 196
  severe 194
ES editing 210
escape sequences 36
etime_ service and utility subprogram 647
exclusive disjunction, logical 97
executable program 141
executing data transfer statements 188
execution environment routines
  OpenMP 667
execution_part 152
execution sequence 25
EXIST specifier, of INQUIRE statement 332
exit_ service and utility subprogram 647
EXIT statement 308
EXP
  intrinsic function 557
  specific name 557
explicit
  interface 144
  typing 52
explicit-shape arrays 66
EXPONENT intrinsic function 558
exponentiation arithmetic operator 91
expressions
  arithmetic 91
  character 93
  constant 87
  dimension bound 63
  general 90
  in FORMAT statement 320
  initialization 87
  logical 96
  primary 99
  relational 94
  restricted 88
  specification 88
  subscript 74
extended
  intrinsic operations 99
  precision (Q) editing 207
external
  files 184
  function 320
  subprograms in the XL Fortran library 639
EXTERNAL attribute 310

F

F (real without exponent) editing 211
f_maketime function 679
f_pthread 677
f_pthread_attr_destroy function 679
f_pthread_attr_getdetachstate function 680
f_pthread_attr_getguardsize function 680
f_pthread_attr_getinheritsched function 681
f_pthread_attr_getschedparam function 681
f_pthread_attr_getschedpolicy function 682
f_pthread_attr_getscope function 682
f_pthread_attr_getstackaddr function 683
f_pthread_attr_getstacksize function 683
f_pthread_attr_init function 684
f_pthread_attr_setdetachstate function 684
f_pthread_attr_setguardsize 685
f_pthread_attr_setinheritsched function 686
f_pthread_attr_setschedparam function 687
f_pthread_attr_setschedpolicy function 688
f_pthread_attr_setscope function 689
f_pthread_attr_setstackaddr function 689
f_pthread_attr_setstacksize function 690
f_pthread_attr_t function 690
f_pthread_cancel function 690
f_pthread_cleanup_pop function 691
f_pthread_cleanup_push function 692
f_pthread_cond_broadcast function 693
f_pthread_cond_destroy function 693
f_pthread_cond_init function 694
f_pthread_cond_signal function 694
f_pthread_cond_t function 695
f_pthread_cond_timedwait function 695
f_pthread_cond_wait function 696
f_pthread_condattr_destroy function 696
f_pthread_condattr_getpshared function 697

f_pthread_condattr_setpshared
function 698
f_pthread_condattr_t function 698
f_pthread_create function 699
f_pthread_detach function 700
f_pthread_equal function 700
f_pthread_exit function 701
f_pthread_getconcurrency
function 701
f_pthread_getschedparam
function 702
f_pthread_getspecific function 702
f_pthread_join function 703
f_pthread_key_create function 704
f_pthread_key_delete function 704
f_pthread_key_t function 705
f_pthread_kill function 705
f_pthread_mutex_destroy
function 705
f_pthread_mutex_getprioceiling
function 706
f_pthread_mutex_init function 706
f_pthread_mutex_lock function 707
f_pthread_mutex_setprioceiling
function 707
f_pthread_mutex_t function 707
f_pthread_mutex_trylock
function 708
f_pthread_mutex_unlock
function 708
f_pthread_mutexattr_destroy
function 709
f_pthread_mutexattr_getprioceiling
function 709
f_pthread_mutexattr_getprotocol
function 709
f_pthread_mutexattr_getpshared
function 710
f_pthread_mutexattr_gettype
function 711
f_pthread_mutexattr_init
function 712
f_pthread_mutexattr_setprioceiling
function 712
f_pthread_mutexattr_setprotocol
function 713
f_pthread_mutexattr_setpshared
function 713
f_pthread_mutexattr_settype
function 714
f_pthread_mutexattr_t function 715
f_pthread_once function 715
f_pthread_once_t function 715
f_pthread_rwlock_destroy
function 715
f_pthread_rwlock_init function 716
f_pthread_rwlock_rdlock
function 717
f_pthread_rwlock_t function 717
f_pthread_rwlock_tryrdlock
function 718
f_pthread_rwlock_trywrlock
function 718
f_pthread_rwlock_unlock
function 719
f_pthread_rwlock_wrlock
function 719
f_pthread_rwlockattr_destroy
function 720
f_pthread_rwlockattr_getpshared
function 720
f_pthread_rwlockattr_init
function 721
f_pthread_rwlockattr_setpshared
function 721
f_pthread_rwlockattr_t function 722
f_pthread_self function 722
f_pthread_setcancelstate
function 722
f_pthread_setcanceltype
function 723
f_pthread_setconcurrency
function 723
f_pthread_setschedparam
function 724
f_pthread_setspecific function 725
f_pthread_t function 725
f_pthread_testcancel function 725
f_sched_param function 726
f_sched_yield function 726
f_timespec function 726
factor
  arithmetic 91
  logical 96
fdate_ service and utility
subprogram 647
fexcp.h include file 620
field editing 204
file position
  BACKSPACE statement, after
  execution 252
  before and after data
  transfer 191
  ENDFILE statement, after
  execution 303
  REWIND statement, after
  execution 387
file positioning statement
  BACKSPACE statement 251
  ENDFILE statement 302
file positioning statement *(continued)*
  REWIND statement 387
FILE specifier
  of INQUIRE statement 332
  of OPEN statement 355
files, external 184
fiosetup_ service and utility
subprogram 648
FIRSTPRIVATE clause 437
fixed source form 17
FLOAT specific name 612
FLOOR intrinsic function 558
flush_ service and utility
subprogram 649
FLUSH compiler directive 462
FMADD intrinsic function 559
FMSUB intrinsic function 559
FMT specifier
  of PRINT statement 368
  of READ statement 375
  of WRITE statement 416
FNMADD intrinsic function 560
FNMSUB intrinsic function 560
FORALL
  construct 113
  statement 311
FORALL (Construct) statement 314
FORM specifier
  of INQUIRE statement 332
  of OPEN statement 355
format
  codes 204
  conditional compilation 22
  control 226
  fixed source form 17
  format-directed formatting 201
  free source form 19
  IBM free source form 21
  specification
    character 319
    interaction with input/output
    list 226
  statement (FORMAT) 315
formatted
  records 183
  specifier of INQUIRE statement
  (FORMATTED) 332
formatting
  description 201
  format-directed 201
  list-directed 227
  namelist 231
fp_trap service and utility
subprogram 650

fpgets and fpsets service and utility subprograms 650
fpscr constants
  Exception Details Flags 641
  Exception Summary Flags 641
  fp_trap constants 642
  general 641
  IEEE Exception Enable Flags 641
  IEEE Exception Status Flags 641
  IEEE Rounding Modes 641
  list 641
fpscr procedures
  clr_fpscr_flags 645
  discussion 639
  fp_trap 650
  get_fpscr 652
  get_fpscr_flags 653
  get_round_mode 654
  set_fpscr 662
  set_fpscr_flags 662
  set_round_mode 663
FRACTION intrinsic function 562
FRE intrinsic function 561
free source form 19
free source form format
  IBM 21
FRSQRTE intrinsic function 561
FSEL intrinsic function 562
ftell_ and ftell64_ service and utility subprograms 651
function
  intrinsic 639
  reference 159
  specification 89
  statement 392
  subprogram 158
  value 159
FUNCTION statement 320

G

G (general) editing 212
GAMMA
  intrinsic function 563
  specific name 563
general expression 90
general service and utility procedures 642
get_fpscr_flags service and utility subprogram 653
get_fpscr service and utility subprogram 652
get_round_mode service and utility subprogram 654
getarg service and utility subprogram 651
getcwd_ service and utility subprogram 652
GETENV intrinsic subroutine 563
getfd service and utility subprogram 652
getgid_ service and utility subprogram 653
getlog_ service and utility subprogram 653
getpid_ service and utility subprogram 654
getuid_ service and utility subprogram 654
global entities 134
global_timef service and utility subprogram 654
gmtime_ service and utility subprogram 655
GO TO statement
  assigned 324
  computed 325
  unconditional 326
GUIDED scheduling type of SCHEDULE directive 491

H

H editing 223
hexadecimal
  (Z) editing 219
  constants 46
HFIX elemental function 564
HFIX specific name 565
Hollerith constants 14, 48
host
  association 133, 137
  scoping unit 133
hostnm_ service and utility subprogram 655
HUGE intrinsic function 565

I

I (integer) editing 214
IABS specific name 525
IACHAR intrinsic function 565
IAND
  intrinsic function 566
  specific name 567
iargc service and utility subprogram 656
IBCLR
  intrinsic function 567
  specific name 567
IBITS
  intrinsic function 567
  specific name 568
IBM free source form 21
IBSET
  intrinsic function 568
  specific name 568
ICHAR
  intrinsic function 569
  specific name 569
ID specifier
  of READ statement 375
  of WAIT statement 413
  of WRITE statement 416
idate_ service and utility subprogram 656
identity arithmetic operator 91
IDIM specific name 552
IDINT specific name 572
IDNINT specific name 599
IEOR
  intrinsic function 569
  specific name 570
ierrno_ service and utility subprogram 656
IF
  construct 121
  statement
    arithmetic 327
    block 328
    logical 329
IF clause 476, 480, 484
IFIX specific name 572
ILEN intrinsic function 570
IMAG
  initializing expressions 88
  intrinsic function 570
implicit
  connection 187
  interface 144
  typing 52
IMPLICIT
  description 329
  statement, storage class assignment and 60
  type determination and 52
implied-DO
  array constructor list in 81
  DATA statement and 278
INCLUDE compiler directive 464
inclusive disjunction, logical 97
incrementation processing 129
INDEPENDENT compiler directive
  discussion 466

INDEX
- initializing expressions 88
- intrinsic function 570
- specific name 571

infinity
- how indicated with numeric output editing 208

inherited length
- by a named constant 264, 407

initial
- line 16
- value, declaring 277

initialization expressions 87
inline comments 16
input/output conditions 193
INQUIRE statement 332
inquiry intrinsic functions 519
- BIT_SIZE 537
- DIGITS 551
- EPSILON 555
- HUGE 565
- KIND 574
- LEN 576
- LOC 581
- MAXEXPONENT 587
- MINEXPONENT 592
- PRECISION 603
- PRESENT 604
- RADIX 608
- RANGE 611
- TINY 632

INT
- initializing expressions 88
- intrinsic function 571
- specific name 572

integer
- data type 29
- editing (I) 214
- pointer association 140
- POINTER statement 366

INTEGER type declaration statement 338
INTENT attribute 342
interaction between input/output list and format specification 226
interface
- blocks 8, 144
- implicit 144
- statement (INTERFACE) 344

interference 446, 466
interlanguage calls
- %VAL and %REF functions 165

internal
- function 320
- procedures 141

intrinsic
- assignment 103
- attribute (INTRINSIC) 345
- data types 29
- functions 519
  - conditional vector merge 546
  - detailed descriptions 523
  - generic 160
  - specific 160
- inquiry 519
- procedures 160
  - description 524
  - discussion 519
  - elemental 519
  - inquiry 519, 520
  - name in an INTRINSIC statement 346
  - subroutines 521
  - transformational 520
- statement (INTRINSIC) 149
- subroutines 521

invocation commands 16
IOR
- intrinsic function 572
- specific name 573

IOSTAT specifier
- of BACKSPACE statement 251
- of CLOSE statement 265
- of ENDFILE statement 302
- of INQUIRE statement 332
- of OPEN statement 355
- of READ statement 375
- of REWIND statement 387
- of WAIT statement 413
- of WRITE statement 416

IOSTAT values 193
IQINT specific name 572
IQNINT specific name 599
irand service and utility subprogram 656
irtc service and utility subprogram 657
ISHFT
- intrinsic function 573
- specific name 573

ISHFTC
- intrinsic function 574
- specific name 574

ISIGN specific name 620
iteration count
- DO statement and 127
- in implied-DO list of a DATA statement 279

itime_ service and utility subprogram 657

J

jdate service and utility subprogram 657

K

keywords
- argument 162
- statement 15

KIND
- intrinsic, constant expressions and 87
- intrinsic, restricted expressions 88
- intrinsic function 574

kind type parameter 27

L

L (logical) editing 215
labels, statement 15
LANGLVL run-time option 199, 234
LASTPRIVATE clause 437
LBOUND array intrinsic function 575
LEADZ intrinsic function 576
LEN
- intrinsic, constant expressions and 87
- intrinsic, restricted expressions 88
- intrinsic function 576
- specific name 577

LEN_TRIM intrinsic function 577
lenchr_ service and utility subprogram 658
length, inherited by a named constant 264, 407
length type parameter 27
letters, character 13
lexical
- tokens 14

lexical extent, definition of 755
LGAMMA
- intrinsic function 577
- specific name 578

LGE
- intrinsic function 578
- specific name 579

LGT
- intrinsic function 579
- specific name 579

library subprograms 639
line breaks, preventing with $ editing 221
lines
- comment 16

lines *(continued)*
conditional compilation 22
continuation 16
debug 16, 18
directive 16, 423
initial 16
source formats and 16
linker options
-brename 642
list-directed formatting 227
literal storage class 59
LLE
intrinsic function 579
specific name 580
LLT
intrinsic function 580
specific name 581
lnblnk_ service and utility subprogram 658
LOC
intrinsic function 118, 581
local entities 134
lock routines
OpenMP 667
LOG intrinsic function 581
LOG10 intrinsic function 582
logical
(L) editing 215
conjunction 97
data type 34
equivalence 97
exclusive disjunction 97
expressions 96
IF statement 329
inclusive disjunction 97
intrinsic function (LOGICAL) 583
negation 97
nonequivalence 97
type declaration statement (LOGICAL) 347
loop
carried dependency 446, 466
control processing 128
DO construct and 126
LSHIFT
elemental function 583
specific name 584
ltime_ service and utility subprogram 658

M

macro, _OPENMP C preprocessor 24
main program 152, 372
many-one section 78
masked array assignment 108
masked ELSEWHERE statement 106, 295
MASTER compiler directive 472
MATMUL array intrinsic function 584
MAX
initializing expressions 88
intrinsic function 586
MAX0 specific name 587
MAX1 specific name 587
MAXEXPONENT intrinsic function 587
MAXLOC array intrinsic function 588
MAXVAL array intrinsic function 589
mclock service and utility subprogram 659
MERGE array intrinsic function 590
MIN
initializing expressions 88
intrinsic function 591
MIN0 specific name 592
MIN1 specific name 592
MINEXPONENT intrinsic function 592
MINLOC array intrinsic function 593
MINVAL array intrinsic function 594
MOD
initializing expressions 88
intrinsic function 596
specific name 596
module
description 8, 153
reference 139, 408
statement (MODULE) 350
MODULE PROCEDURE statement 351
MODULO intrinsic function 596
multibyte characters 37
multiplication arithmetic operator 91
MVBITS intrinsic subroutine 597

N

name
common block 267
description 14
determining storage class of 59
determining type of 52
entry 304
name *(continued)*
of a generic or specific function 160
scope of a 134
NAME specifier, of INQUIRE statement 332
named common block 268
NAMED specifier, of INQUIRE statement 332
namelist
formatting 231
group 14
NAMELIST
run-time option 236
statement 353
NEAREST intrinsic function 598
negation
arithmetic operator 91
logical operator 97
NEQV logical operator 97
NEXTREC specifier
of INQUIRE statement 332
NINT
initializing expressions 88
intrinsic function 598
specific name 599
NML specifier
of READ statement 375
of WRITE statement 416
nonequivalence, logical 97
NOT
intrinsic function 599
logical operator 97
specific name 599
NULL
initializing expressions 88
intrinsic function 599
NULLIFY statement 354
NUM_PARTHDS inquiry intrinsic function 601
NUM specifier
of READ statement 375
of WRITE statement 416
NUM_USRTHDS inquiry intrinsic function 602
NUMBER_OF_PROCESSORS intrinsic function 601
NUMBER specifier, of INQUIRE statement 332
numeric edit descriptors 204

O

O (octal) editing 216
objects, data 28
octal (O) editing 216

octal constants 47
omp_destroy_lock OpenMP lock routine 668
omp_get_dynamic execution environment routine 668
omp_get_max_threads execution environment routine 668
omp_get_nested execution environment routine 669
omp_get_num_procs execution environment routine 669
omp_get_num_threads execution environment routine 669
omp_get_thread_num execution environment routine 670
omp_in_parallel execution environment routine 671
omp_init_lock lock routine 672
omp_set_dynamic execution environment routine 672
omp_set_lock lock routine 673
omp_set_nested execution environment routine 674
omp_set_num_threads execution environment routine 674
omp_test_lock lock routine 675
omp_unset_lock lock routine 675
ONLY clause of USE statement 408
OPEN statement 355
OPENED specifier, of INQUIRE statement 332
OpenMP
  execution environment routines
    description 667
    omp_get_dynamic 668
    omp_get_max_threads 668
    omp_get_nested 669
    omp_get_num_procs 669
    omp_get_num_threads 669
    omp_get_thread_num 670
    omp_in_parallel 671
    omp_set_dynamic 672
    omp_set_nested 674
    omp_set_num_threads 674
  lock routines
    description 667
    omp_destroy_lock 668
    omp_init_lock 672
    omp_set_lock 673
    omp_test_lock 675
    omp_unset_lock 675
operations
  defined 99
  extended intrinsic 99
operators
  arithmetic 91
  character 94
  defined 150
  logical 97
  precedence of 100
  relational 94
optional arguments 167
OPTIONAL attribute 361
OR
  logical operator 97
  specific name 573
order
  of elements in an array 74
  of statements 25
ORDERED clause of DO directive 456
ORDERED clause of PARALLEL DO directive 480
ORDERED compiler directive 473

P

P (scale factor) editing 224
PACK array intrinsic function 602
PAD specifier
  of INQUIRE statement 332
  of OPEN statement 355
PARALLEL compiler directive
  discussion 476
PARALLEL DO compiler directive
  discussion 479
  ORDERED clause 480
  SCHEDULE clause 479
PARALLEL SECTIONS compiler directive
  discussion 483
PARAMETER attribute 362
PAUSE statement 364
pending control mask 108
PERMUTATION compiler directive 487
pointee
  arrays 67
  POINTER statement and 367
pointer
  assignment 117
  association 139
  attribute, POINTER (Fortran 90) 364
POSITION specifier
  of INQUIRE statement 332
  of OPEN statement 355
positional (T, TL, TR, and X) editing 225
precedence
  of all operators 100
  of arithmetic operators 91
  of logical operators 97
PRECISION intrinsic function 603
precision of real objects 31
preconnection 186
PREFETCH_BY_LOAD compiler directive 488
PREFETCH_FOR_LOAD compiler directive 488
PREFETCH_FOR_STORE compiler directive 488
PRESENT intrinsic function 361, 604
primaries (expressions) 86
primary expressions 99
PRINT statement 368
PRIVATE
  attribute 370
  statement 39, 370
PRIVATE clause 436
procedure
  dummy 171
  external 141, 372
  internal 141
procedure, invoked by a subprogram 141
procedure references 159
PROCESSORS_SHAPE intrinsic function 605
PRODUCT array intrinsic function 605
PROGRAM statement 372
program unit 141
Pthreads Library, AIX 677
Pthreads Library Module
  descriptions of functions in 677
  f_maketime function 679
  f_pthread_attr_destroy function 679
  f_pthread_attr_getdetachstate function 680
  f_pthread_attr_getguardsize function 680
  f_pthread_attr_getinheritsched function 681
  f_pthread_attr_getschedparam function 681
  f_pthread_attr_getschedpolicy function 682
  f_pthread_attr_getscope function 682
  f_pthread_attr_getstackaddr 683

Pthreads Library Module *(continued)*
f_pthread_attr_getstacksize function 683
f_pthread_attr_init function 684
f_pthread_attr_setdetachstate function 684
f_pthread_attr_setguardsize function 685
f_pthread_attr_setinheritsched function 686
f_pthread_attr_setschedparam function 687
f_pthread_attr_setschedpolicy function 688
f_pthread_attr_setscope function 689
f_pthread_attr_setstackaddr function 689
f_pthread_attr_setstacksize function 690
f_pthread_attr_t function 690
f_pthread_cancel function 690
f_pthread_cleanup_pop function 691
f_pthread_cleanup_push function 692
f_pthread_cond_broadcast function 693
f_pthread_cond_destroy function 693
f_pthread_cond_init function 694
f_pthread_cond_signal function 694
f_pthread_cond_t function 695
f_pthread_cond_timedwait function 695
f_pthread_cond_wait function 696
f_pthread_condattr_destroy function 696
f_pthread_condattr_getpshared function 697
f_pthread_condattr_setpshared function 698
f_pthread_condattr_t function 698
f_pthread_create function 699
f_pthread_detach function 700
f_pthread_equal function 700
f_pthread_exit function 701
f_pthread_getconcurrency function 701
f_pthread_getschedparam function 702
Pthreads Library Module *(continued)*
f_pthread_getspecific function 702
f_pthread_join function 703
f_pthread_key_create function 704
f_pthread_key_delete function 704
f_pthread_key_t function 705
f_pthread_kill function 705
f_pthread_mutex_destroy function 705
f_pthread_mutex_getprioceiling function 706
f_pthread_mutex_init function 706
f_pthread_mutex_lock function 707
f_pthread_mutex_setprioceiling function 707
f_pthread_mutex_t function 707
f_pthread_mutex_trylock function 708
f_pthread_mutex_unlock function 708
f_pthread_mutexattr_destroy function 709
f_pthread_mutexattr_getprioceiling function 709
f_pthread_mutexattr_getprotocpol function 709
f_pthread_mutexattr_getpshared function 710
f_pthread_mutexattr_gettype function 711
f_pthread_mutexattr_init function 712
f_pthread_mutexattr_setprioceiling function 712
f_pthread_mutexattr_setprotocol function 713
f_pthread_mutexattr_setpshared function 713
f_pthread_mutexattr_settype function 714
f_pthread_mutexattr_t function 715
f_pthread_once function 715
f_pthread_once_t function 715
f_pthread_rwlock_destroy function 715
f_pthread_rwlock_init function 716
f_pthread_rwlock_rdlock function 717
Pthreads Library Module *(continued)*
f_pthread_rwlock_t function 717
f_pthread_rwlock_tryrdlock function 718
f_pthread_rwlock_trywrlock function 718
f_pthread_rwlock_unlock function 719
f_pthread_rwlock_wrlock function 719
f_pthread_rwlockattr_destroy function 720
f_pthread_rwlockattr_getpshared function 720
f_pthread_rwlockattr_init function 721
f_pthread_rwlockattr_setpshared function 721
f_pthread_rwlockattr_t function 722
f_pthread_self function 722
f_pthread_setcancelstate function 722
f_pthread_setcanceltype function 723
f_pthread_setchedparam function 724
f_pthread_setconcurrency function 723
f_pthread_setspecific function 725
f_pthread_t function 725
f_pthread_testcancel function 725
f_sched_param function 726
f_sched_yield function 726
f_timespec function 726
PUBLIC attribute 373
PURE 176
pure procedures 176

Q

Q (extended precision) editing 207
QABS specific name 525
QACOS specific name 526
QACOSD specific name 527
QARCOS specific name 526
QARSIN specific name 532
QASIN specific name 532
QASIND specific name 533
QATAN specific name 534
QATAN2 specific name 536
QATAN2D specific name 537
QATAND specific name 535

QCMPLX
initializing expressions 88
intrinsic function 607
specific name 607
QCONJG specific name 541
QCOS specific name 541
QCOSD specific name 542
QCOSH specific name 542
QDIM specific name 552
QERF specific name 556
QERFC specific name 557
QEXP specific name 557
QEXT
initializing expressions 88
intrinsic function 607
specific name 608
QEXTD specific name 608
QFLOAT specific name 608
QGAMMA specific name 563
QIMAG specific name 528
QINT specific name 529
QLGAMA specific name 578
QLOG specific name 582
QLOG10 specific name 583
QMAX1 specific name 587
QMIN1 specific name 592
QMOD specific name 596
QNINT specific name 531
QPROD specific name 553
QREAL specific name 612
QSIGN specific name 620
QSIN specific name 621
QSIND specific name 622
QSINH specific name 622
qsort_ service and utility subprogram 659
qsort_down service and utility subprogram 660
qsort_up service and utility subprogram 661
QSQRT specific name 625
QTAN specific name 630
QTAND specific name 631
QTANH specific name 631

R

RADIX intrinsic function 608
RAND intrinsic function 608
RANDOM_NUMBER intrinsic subroutine 609
RANDOM_SEED intrinsic subroutine 609
RANGE intrinsic function 611
rank
of array sections 80
rank *(continued)*
of arrays 64
READ
specifier, of INQUIRE statement 332
statement 374
READWRITE specifier, of INQUIRE statement 332
REAL
initializing expressions 88
intrinsic function 612
specific name 612
real data type 30
real editing
E (with exponent) 207
F (without exponent) 211
G (general) 212
REAL type declaration statement 381
REC specifier
of READ statement 375
of WRITE statement 416
RECL specifier
of INQUIRE statement 332
of OPEN statement 355
records
description 183
endfile 184
formatted 183
unformatted 184
recursion
ENTRY statement and 305
FUNCTION statement and 322
procedures and 175
SUBROUTINE statement and 397
RECURSIVE keyword 322, 397
REDUCTION clause 438
reference, function 159
relational
expressions 94
operators 94
REPEAT
intrinsic function 88, 613
intrinsic initialization expressions 87
repeat specification 316
RESHAPE
array intrinsic function 88, 613
array intrinsic initialization expressions 87
restricted expression 88
RESULT keyword 304, 321
result variable 304, 321
return points and specifiers, alternate 161
return specifier 25
RETURN statement 385
REWIND statement 387
right margin 17
rounding mode 92
RRSPACING intrinsic function 614
RSHIFT
elemental function 615
specific name 615
rtc service and utility subprogram 661
run-time options
changing with SETRTEOPTS procedure 663
CNVERR
conversion errors and 198
READ statement and 380
WRITE statement and 420
ERR_RECOVERY 199
BACKSPACE statement and 252
conversion errors and 198
ENDFILE statement and 303
OPEN statement and 360
READ statement and 380
REWIND statement and 388
severe errors and 194
WRITE statement and 420
LANGLVL 199, 234
NAMELIST 236
NLWIDTH 237
UNIT_VARS 187, 355
RUNTIME scheduling type of SCHEDULE directive 491

S

S (sign control) editing 224
SAVE attribute 388
scalar-int-constant-name 27
scale factor (P) editing 224
SCALE intrinsic function 615
SCAN
initializing expressions 88
intrinsic function 616
SCHEDULE clause
of DO (work-sharing) directive 455
SCHEDULE clause, of PARALLEL DO directive 479
SCHEDULE compiler directive
AFFINITY scheduling type 491
discussion 491
DYNAMIC scheduling type 491

SCHEDULE compiler directive (continued)
  GUIDED scheduling type 491
  RUNTIME scheduling type 491
  STATIC scheduling type 491
scheduling, block cyclic 493
scope
  data scope attribute clauses 430
scope, entities and 134
scoping unit 133
section_subscript, syntax of for array section 75
SECTIONS compiler directive
  discussion 498
SELECT CASE statement
  CASE construct 123
  CASE statement and 258
  description 390
SELECTED_INT_KIND
  intrinsic function 88, 616
  intrinsic initialization expressions 87
SELECTED_REAL_KIND
  intrinsic function 88, 617
  intrinsic initialization expressions 87
selector 14
semicolon statement separator 18, 20
sequence derived type 40
SEQUENCE statement 39, 391
SEQUENTIAL specifier, of INQUIRE statement 332
service and utility subprograms
  alarm_ 644
  bic_ 644
  bis_ 645
  bit_ 645
  clock_ 645
  clr_fpscr_flags 645
  ctime_ 646
  date 646
  discussion 639
  dtime_ 646
  efficient floating-point control and inquiry procedures 639
  etime_ 647
  exit_ 647
  fdate_ 647
  fiosetup_ 648
  flush_ 649
  fp_trap 650
  fpgets and fpsets 650
  ftell_ and ftell64_ 651
  general 642
service and utility subprograms (continued)
  get_fpscr 652
  get_fpscr_flags 653
  get_round_mode 654
  getarg 651
  getcwd_ 652
  getfd 652
  getgid_ 653
  getlog_ 653
  getpid_ 654
  getuid_ 654
  global_timef 654
  gmtime_ 655
  hostnm_ 655
  iargc 656
  idate_ 656
  ierrno_ 656
  irand 656
  irtc 657
  itime_ 657
  jdate 657
  lenchr_ 658
  lnblnk_ 658
  ltime_ 658
  mclock 659
  qsort_ 659
  qsort_down 660
  qsort_up 661
  rtc 661
  set_fpscr 662
  set_fpscr_flags 662
  set_round_mode 663
  setrteopts 663
  sleep_ 664
  time_ 664
  timef 664
  timef_delta 664
  umask_ 665
  usleep_ 665
  xl_ _trbk 665
SET_EXPONENT intrinsic function 618
set_fpscr_flags service and utility subprogram 662
set_fpscr service and utility subprogram 662
set_round_mode service and utility subprogram 663
setrteopts service and utility subprogram 663
shape
  array intrinsic function (SHAPE) 618
  of an array 64
shape (continued)
  of array sections 80
SHARED clause 443
SIGN
  initializing expressions 88
  intrinsic function 619
  specific name 620
sign control (S, SS, and SP) editing 224
signal.h include file 620
SIGNAL intrinsic subroutine 620
SIN
  intrinsic function 621
  specific name 621
SIND
  intrinsic function 622
  specific name 622
SINGLE / END SINGLE compiler directive 502
SINH
  intrinsic function 622
  specific name 622
SIZE
  array intrinsic function 623
  specifier, of READ statement 375
slash (/) editing 220
sleep_ service and utility subprogram 664
SMP
  concepts 423
  directives 423
SNGL specific name 612
SNGLQ specific name 612
sorting (qsort_ procedure) 659
source file options 470, 490
source formats
  conditional compilation 22
  fixed source form 17
  free source form 19
  IBM free source form 21
SOURCEFORM compiler directive 506
SP (sign control) editing 224
SPACING intrinsic function 623
special characters 13
specification array 65
specification expression 88
specification function 89
specification_part 152
SPREAD array intrinsic function 624
SQRT
  intrinsic function 625
  specific name 625

SRAND intrinsic subroutine 626
SS (sign control) editing 224
statements
  assignment 103
  block 121
  description 15
  discussion 241
  entities 134, 136
  function statement 392
  label assignment (ASSIGN) statement 248
  labels 15
  order 25
  terminal 126
STATIC
  attribute 394
  scheduling type of SCHEDULE directive 491
STATUS specifier
  of CLOSE statement 265
  of OPEN statement 355
STOP statement 395
storage
  classes for variables
    description 59
    fundamental 59
    literal 59
    secondary 59
  sequence within common blocks 269
  sharing
    using common blocks 268
    using EQUIVALENCE 306
    using integer pointers 140
    using pointers 139
structure
  array components 79
  components 41
  constructor 44
  description 41
  scalar components 43
subobjects of variables 28
subprograms
  external 141
  function 320
    external 158
    internal 158
  internal 141
  invocation 141
  references 159
  service and utility 639
  subroutine 158
subroutine
  functions and 157
  intrinsic 639
subroutine *(continued)*
  statement (SUBROUTINE) 396
subscript_triplet, syntax of 76
subscripts 74
substring
  character 37
  ranges
    relationship to array sections 78
    specifying 75
subtraction arithmetic operator 91
SUM array intrinsic function 626
symmetric multiprocessing
  concepts 423
  directives 423
SYSTEM_CLOCK intrinsic subroutine 629
system inquiry intrinsic functions 520
SYSTEM intrinsic subroutine 628

T

T (positional) editing 225
tabs, formatting 17
TAN
  intrinsic function 630
  specific name 630
TAND
  intrinsic function 631
  specific name 631
TANH
  intrinsic function 631
  specific name 631
TARGET attribute 398
terminal statement 126
thread-safing
  of Fortran 90 pointers 365
  pthreads library module 677
thread visible variables 462
THREADLOCAL compiler directive 507
THREADPRIVATE compiler directive 510
time_ service and utility subprogram 664
time zone, setting 547
timef_delta service and utility subprogram 664
timef service and utility subprogram 664
TINY intrinsic function 632
TL (positional) editing 225
TR (positional) editing 225
TRANSFER intrinsic function
  description 632
TRANSFER intrinsic function *(continued)*
  initialization expressions 87
  restricted expressions 88
transfer of control
  description 25
  in a DO loop 128
TRANSFER specifier, of INQUIRE statement 332
transformational intrinsic functions 520
TRANSPOSE array intrinsic function 633
TRIM intrinsic function
  description 634
  initialization expressions 87
  restricted expressions 88
type, determining 52
type declaration 402
  BYTE 253
  CHARACTER 260
  COMPLEX 270
  DOUBLE COMPLEX 287
  DOUBLE PRECISION 290
  INTEGER 338
  LOGICAL 347
  REAL 381
  TYPE 399
type parameters and specifiers 27
typeless constants
  binary 48
  hexadecimal 46
  Hollerith 48
  octal 47
  using 49
TZ environment variable 547

U

UBOUND array intrinsic function 634
umask_ service and utility subprogram 665
unambiguous references 147
unary operations 85
unconditional GO TO statement 326
unformatted records 184
UNFORMATTED specifier
  of INQUIRE statement 332
Unicode characters and filenames
  and character constants 221
  character constants and 37
  compiler option for 37
  environment variable for 37
  H editing and 223

Unicode characters and filenames *(continued)*
- Hollerith constants and 49

UNIT specifier
- of BACKSPACE statement 251
- of CLOSE statement 265
- of ENDFILE statement 302
- of INQUIRE statement 332
- of OPEN statement 355
- of READ statement 375
- of REWIND statement 387
- of WRITE statement 416

units, external files reference 186
UNPACK array intrinsic function 635
UNROLL compiler directive 514
use association 139, 408
USE statement 408
usleep_ service and utility subprogram 665

V

value separators 227
variable
- description 28
- format expressions and 320

vector subscripts 78
VERIFY
- initializing expressions 88
- intrinsic function 636

VIRTUAL statement 410
VOLATILE attribute 410

W

WAIT statement 412
WHERE
- construct 106
- construct statement 414
- nested in FORALL 115
- statement 106, 414

where_construct_name 106, 295, 298, 414
white space 13
whole array 63
work-sharing constructs
- DO / END DO compiler directives 455
- SECTIONS / END SECTIONS compiler directives 498
- SINGLE / END SINGLE compiler directives 502

WRITE
- specifier of INQUIRE statement 332
- statement 416

X

X (positional) editing 225
xl_ _trbk service and utility subprogram 665
xlf_fp_util module 639
xlfutility module 642
XOR
- logical operator 97
- specific name 570

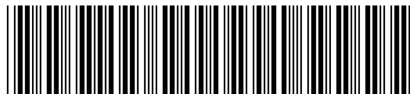
Z

Z (hexadecimal) editing 219
ZABS specific name 525
ZCOS specific name 541
zero-length string 37
zero-sized array 63
ZEXP specific name 557
ZLOG specific name 582
ZSIN specific name 621
ZSQRT specific name 625

Part Number: CT763NA

Printed in the United States of America

SC09-2867-00



CT763NA