IBM XL Fortran Advanced Edition V10.1 for Linux

Language Reference
First Edition (November 2005)

This edition applies to Version 10.1 of IBM XL Fortran Advanced Edition V10.1 for Linux (Program 5724-M17) 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.

IBM welcomes your comments. You can send your comments electronically to the network ID listed below. Be sure to include your entire network address if you wish a reply.

compinfo@ca.ibm.com

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, 2005. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
# Contents

## About this document [. . . . . . . . . . xi]
- Who should read this document ........ xi
- How to use this document ........ xii
- How this document is organized ....... xi

## Chapter 1. XL Fortran for Linux .......... 1
- Language standards .................. 1
  - Fortran 2003 Standard ............... 1
  - Fortran 95 ......................... 1
  - Fortran 90 ......................... 2
  - IBM Extensions .................... 2
- Other standards and standards documents .... 2

## Chapter 2. Fundamentals of the XL Fortran language .......... 3
- Characters ......................... 3
- Names .......................... 4
- Statements ....................... 5
  - Statement keywords ........... 5
  - Statement labels ............... 5
- Lines and source formats ....... 5
  - Fixed source form ....... 6
  - Free source form ........ 9
  - IBM free source form .... 11
- Conditional compilation .... 11
- Order of statements and execution sequence .... 14

## Chapter 3. Data types and data objects .......... 15
- Data types ........................ 15
  - Type parameters and specifiers ........ 15
  - Data objects ..................... 15
  - Constants ....................... 16
  - Automatic objects .............. 16
- Intrinsic types ................ 16
  - Integer ........................ 16
  - Real .......................... 18
  - Vector ........................ 20
  - Pixel .......................... 21
  - Unsigned ....................... 21
  - Complex ....................... 22
  - Logical ....................... 24

## Chapter 4. Array concepts .......... 63
- 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 .......... 67
  - Examples of assumed-shape arrays ..... 68
- Deferred-shape arrays ...... 68
  - Allocatable arrays .......... 69
  - Array pointers .............. 70
- Assumed-size arrays .......... 70
  - Examples of assumed-size arrays ..... 72
- Array elements .............. 72
  - Notes ........................ 73
  - Array element order ............ 73
- Array sections ............... 73
  - Subscript triplets ........... 75
  - Vector subscripts .......... 76
  - Array sections and substring ranges .... 77
  - Array sections and structure components ..... 77
  - Rank and shape of array sections .... 79
  - Array constructors .......... 79
  - Implied-DO list for an array constructor .... 80

© Copyright IBM Corp. 1990, 2005
Expressions involving arrays ........................................ 81

Chapter 5. Expressions and assignment ...................................... 83
Introduction to expressions and assignment .......................... 83
  Primary .............................................. 84
Constant expressions ............................................. 84
  Examples of constant expressions .......................... 85
Initialization expressions .......................................... 85
  Examples of initialization expressions .......................... 86
Specification expressions ............................................ 86
  Examples of specification expressions .......................... 87
Operators and expressions ............................................ 88
  Arithmetic ........................................... 88
  Character ............................................ 91
  General ............................................. 91
  Logical ............................................. 92
  Primary ............................................. 94
  Relational .......................................... 95
Extended intrinsic and defined operations ......................... 96
How expressions are evaluated .................................... 97
  Precedence of operators .................................. 97
  Using BYTE data objects ................................ 100
Intrinsic assignment ............................................. 100
  Arithmetic conversion ..................................... 102
WHERE construct ........................................ 103
  Interpreting masked array assignments ...................... 105
FORALL construct ....................................... 109
  Interpreting the FORALL construct ....................... 111
Pointer assignment ........................................... 112
  Examples of pointer assignment ............................ 113
Procedure pointer assignment .................................. 114
Integer pointer assignment ..................................... 115
  Example of integer pointer assignment ...................... 115

Chapter 6. Execution Control ............................................. 117
statement blocks ........................................... 117
ASSOCIATE Construct ....................................... 117
DO construct ........................................... 119
  The terminal statement .................................. 119
DO WHILE construct ......................................... 123
  Example ............................................. 123
IF construct ............................................. 124
  Example ............................................. 125
SELECT CASE construct ..................................... 125
  Examples ............................................ 127
Branching .................................................. 127

Chapter 7. Program units and procedures .................................... 129
Scope .................................................. 129
  The scope of a name .................................. 130
Association .............................................. 134
  Host association ..................................... 134
  Use association ...................................... 135
Construct Association ......................................... 135
  Pointer association .................................... 136
  Integer pointer association .............................. 137
Program units, procedures, and subprograms .................... 138
  Internal procedures .................................... 138
  Interface concepts ...................................... 139
  Interface blocks ....................................... 141
    Example of an interface block .......................... 143
Generic interface blocks ....................................... 144
  Unambiguous generic procedure references .................. 144
  Extending intrinsic procedures with generic interface blocks .......................... 145
  Defined operators ...................................... 146
  Defined assignment ...................................... 147
Main program .............................................. 148
Modules .................................................. 149
  Example of a module ................................... 152
Block data program unit ..................................... 153
  Example of a block data program unit ..................... 154
Function and subroutine subprograms ........................... 154
  Procedure references ................................... 155
Intrinsic procedures ......................................... 156
  Conflicts between intrinsic procedure names and other names .... 157
Arguments .................................................. 157
  Actual argument specification ............................ 157
Argument association ........................................ 160
  %VAL and %REF ........................................ 161
  Intent of dummy arguments ................................ 162
  Optional dummy arguments ................................ 163
  Restrictions on optional dummy arguments not present .... 163
  Length of character arguments ............................ 164
  Variables as dummy arguments ................................ 164
  Allocatable objects as dummy arguments .................... 166
  Pointers as dummy arguments ................................ 167
  Procedures as dummy arguments ................................ 167
  Asterisks as dummy arguments ................................ 168
  Resolution of procedure references ......................... 169
  Rules for resolving procedure references to names .......... 169
  Resolving procedure references to generic names ............ 170
Recursion .................................................. 171
Pure procedures ............................................ 171
  Examples ............................................ 173
Elemental Procedures ....................................... 174
  Examples ............................................ 175

Chapter 8. XL Fortran Input/Output .................................. 177
Records .................................................. 177
  Formatted records ..................................... 177
  Unformatted records .................................... 177
  Endfile records ...................................... 177
Files ................................................... 178
  Definition of an external file ................................ 178
  File access methods .................................... 178
Units .................................................... 180
  Connection of a unit ................................... 180
Data transfer statements ....................................... 182
  Asynchronous Input/Output ................................ 183
  Advancing and nonadvancing Input/Output .................... 184
  File position before and after data transfer .................. 185
Conditions and IOSTAT values ................................ 186
  End-of-record conditions ................................ 187
  End-of-file conditions ................................... 187
  Error conditions ....................................... 188
Chapter 9. Input/Output formatting  195
Format-directed formatting  195
Complex editing  195
Data edit descriptors  195
Control edit descriptors  198
Character string edit descriptors  199
Interaction of Input/Output lists and format specifications  200
Comma-separated Input/Output  201
Data edit descriptors  202
A (Character) Editing  202
B (Binary) Editing  203
E, D, and Q (Extended Precision) Editing  204
EN Editing  206
ES Editing  207
F (Real without Exponent) Editing  208
G (General) Editing  210
H Editing  211
I (Integer) Editing  212
L (Logical) Editing  213
O (Octal) Editing  214
Q (Character Count) Editing  215
Z (Hexadecimal) Editing  217
Control edit descriptors  218
/ (Slash) Editing  218
: (Colon) Editing  219
$ (Dollar) Editing  219
BN (Blank Null) and BZ (Blank Zero) Editing  220
P (Scale Factor) Editing  221
S, SP, and SS (Sign Control) Editing  221
T, TL, TR, and X (Positional) Editing  222
List-directed formatting  223
Value separators  223
List-directed input  223
List-directed output  225
Namelist formatting  226
Namelist input  227
Namelist output  231

Chapter 10. Statements and attributes  235
Attributes  238
ALLOCATABLE  238
ALLOCATE  240
ASSIGN  242
ASSOCIATE  243
AUTOMATIC  244
BACKSPACE  246
BIND  247
BLOCK DATA  248
BYTE  249
CALL  252
CASE  254
CHARACTER  256
CLOSE  260
COMMON  262
COMPLEX  266
CONTAINS  271
CONTINUE  271
CYCLE  272
DATA  273
DEALLOCATE  276
Derived Type  278
DIMENSION  279
DO  280
DO WHILE  282
DOUBLE COMPLEX  283
DOUBLE PRECISION  286
ELSE  289
ELSE IF  290
ELSEWHERE  290
END  292
END (Construct)  293
END INTERFACE  296
END TYPE  297
ENDFILE  298
ENTRY  300
ENUM/END ENUM  303
EQUIVALENCE  305
EXIT  307
EXTERNAL  308
FLUSH  309
FORALL  311
FORALL (construct)  314
FORMAT  315
FUNCTION  317
GO TO (assigned)  321
GO TO (computed)  322
GO TO (unconditional)  323
IF (arithmetic)  324
IF (block)  325
IF (logical)  326
IMPLICIT  326
IMPORT  329
INQUIRE  330
INTEGER  336
INTENT  341
INTERFACE  343
INTRINSIC  344
LOGICAL  346
MODULE  351
MODULE PROCEDURE  352
NAMELIST  353
NULLIFY  354
OPEN  355
OPTIONAL  360
PARAMETER  362
PAUSE  363
POINTER (Fortran 90)  364
POINTER (integer)  366
PRINT  367
PRIVATE  369
PROCEDURE  371
PROGRAM  374
PROTECTED  374
PUBLIC  376
READ  377
REAL  385
RECORD  389
RETURN  390
REWIND  392
SAVE  393
SELECT CASE  395
Chapter 11. Directives .................................. 433
Comment and noncomment form directives 433
  Comment form directives 433
  Noncomment form directives 435
Directives and optimization 436
  Assertive directives 436
  Directive for Loop Optimization 436
Detailed directive descriptions 436
  ASSERT 436
  BLOCK_LOOP 438
  CANCEL 440
  COLLAPSE 441
  EJECT 442
  INCLUDE 442
  INDEPENDENT 444
  LINE 447
  LOOPID 449
  NEW 450
  NOSIMD 451
  NOVECTOR 451
  PERMUTATION 452
  @PROCESS 453
  SNAPSHOTT 454
  SOURCEFORM 455
  STREAM_UNROLL 456
  SUBSCRIPTORDER 458
  UNROLL 460
  UNROLL_AND_FUSE 461

Chapter 12. Hardware-specific directives ........................ 465
  CACHE_ZERO ......................................... 465
  EIEIO ............................................. 465
  ISYNC ............................................. 466
  LIGHT_SYNC ....................................... 466
  PREFETCH ......................................... 466
  PROTECTED STREAM .................................. 469

Chapter 13. Intrinsic procedures .......................... 473
Classes of intrinsic procedures .............................. 473
  Inquiry intrinsic functions 473
  Elemental intrinsic procedures 473
  System inquiry intrinsic functions 474
  Transformational intrinsic functions 474
  Intrinsic subroutines 475
  Data representation models 475
    Integer bit model 475
    Integer data model 476
    Real data model 477
  Detailed descriptions of intrinsic procedures 477
    ABORT() ........................................ 478
    ABS(A) ......................................... 478
    ACHAR(I) ....................................... 479
    ACS(X) ......................................... 480
    ACOSD(X) ....................................... 480
    ADJUSTL(STRING) ................................ 481
    ADJUSTR(STRING) ................................ 481
    AIMAG(Z), IMAG(Z) ................................ 482
    AINT(A, KIND) ................................... 483
    ALIGNED(K,M) .................................... 483
    ALL(MASK, DIM) .................................. 484
    ALLOCATED(X) ................................... 485
    ANINT(A, KIND) .................................. 486
    ANY(MASK, DIM) .................................. 486
    ASIN(X) ......................................... 487
    ASIND(X) ........................................ 488
    ASSOCIATED(POINTER, TARGET) ..................... 488
    ATAN(X) ......................................... 489
    ATAND(X) ....................................... 490
    ATAN2(Y, X) ..................................... 491
    ATAN2D(Y, X) .................................... 492
    BIT_SIZE(I) ..................................... 493
    BETEST(I, POS) .................................. 493
    CEILING(A, KIND) ................................ 494
    CHAR(I, KIND) .................................... 495
    CMPLX(X, Y, KIND) ................................ 496
    COMMAND_ARGUMENT_COUNT() ....................... 497
    CONJG(Z) ....................................... 497
    COS(X) .......................................... 498
    COSD(X) ......................................... 499
    COSH(X) ......................................... 499
    COUNT(MASK, DIM) ................................ 500
    CPU_TIME(TIME) .................................. 501
    CSHIFT(ARRAY, SHIFT, DIM) ......................... 502
    CVMGx(TSOURCE, FSOURCE, MASK) .................... 503
    DATE_AND_TIME(DATE, TIME, ZONE, VALUES) ....... 505
    DBLE(A) .......................................... 506
    DCMPLX(X, Y) .................................... 507
    DIGITS(X) ........................................ 508
    DIM(X, Y) ........................................ 509
    DOT_PRODUCT(VECTOR_A, VECTOR_B) ................. 509
    DPROD(X, Y) ..................................... 510
    EOSHIFT(ARRAY, SHIFT, BOUNDARY, DIM) ............ 511
    EPSILON(X) ...................................... 512
    ERF(X) ........................................... 513
    ERF(C) ........................................... 514
    EXP(X) ........................................... 514
    EXPONENT(X) ..................................... 515
    FLOOR(A, KIND) ................................... 516
    FRACTION(X) ..................................... 517
    GAMMA(X) ........................................ 517
    GETENV(NAME, VALUE) ............................. 518
    GET_COMMAND(COMMAND, LENGTH, STATUS) ........ 519
<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET_COMMAND_ARGUMENT</td>
<td>520</td>
</tr>
<tr>
<td>GET_ENVIRONMENT_VARIABLE</td>
<td>521</td>
</tr>
<tr>
<td>HFIX</td>
<td>522</td>
</tr>
<tr>
<td>IACHAR(C)</td>
<td>523</td>
</tr>
<tr>
<td>LAND(I, J)</td>
<td>524</td>
</tr>
<tr>
<td>IBCLR(I, POS)</td>
<td>525</td>
</tr>
<tr>
<td>IBITS(I, POS, LEN)</td>
<td>526</td>
</tr>
<tr>
<td>IBSET(I, POS)</td>
<td>526</td>
</tr>
<tr>
<td>ICHAR(C)</td>
<td>527</td>
</tr>
<tr>
<td>IEEOR(I, J)</td>
<td>528</td>
</tr>
<tr>
<td>ILEN(I)</td>
<td>529</td>
</tr>
<tr>
<td>IMAG(Z)</td>
<td>529</td>
</tr>
<tr>
<td>INDEX.STRING, SUBSTRING, BACK)</td>
<td>529</td>
</tr>
<tr>
<td>INT(A, KIND)</td>
<td>530</td>
</tr>
<tr>
<td>INT2(A)</td>
<td>531</td>
</tr>
<tr>
<td>IOR(I, J)</td>
<td>532</td>
</tr>
<tr>
<td>ISHFT(I, SHIFT)</td>
<td>533</td>
</tr>
<tr>
<td>ISHFTC(I, SHIFT, SIZE)</td>
<td>534</td>
</tr>
<tr>
<td>KIND(X)</td>
<td>535</td>
</tr>
<tr>
<td>LBNDIARRAY, DIM)</td>
<td>535</td>
</tr>
<tr>
<td>LEADZ(I)</td>
<td>536</td>
</tr>
<tr>
<td>LEN.STRING)</td>
<td>537</td>
</tr>
<tr>
<td>LEN.Trim(STRING)</td>
<td>537</td>
</tr>
<tr>
<td>LGAMMA(X)</td>
<td>538</td>
</tr>
<tr>
<td>LGM1(string_a, string_b)</td>
<td>539</td>
</tr>
<tr>
<td>LGT(string_a, string_b)</td>
<td>539</td>
</tr>
<tr>
<td>LLE(STRING_A, STRING_B)</td>
<td>540</td>
</tr>
<tr>
<td>LIT(STRING_A, STRING_B)</td>
<td>541</td>
</tr>
<tr>
<td>LOC(X)</td>
<td>542</td>
</tr>
<tr>
<td>LOG(X)</td>
<td>542</td>
</tr>
<tr>
<td>LOG10(X)</td>
<td>543</td>
</tr>
<tr>
<td>LOGICAL(L, KIND)</td>
<td>544</td>
</tr>
<tr>
<td>LSHIFT(L, SHIFT)</td>
<td>544</td>
</tr>
<tr>
<td>MATMUL(MATRIX_A, MATRIX_B, MINDIM)</td>
<td>545</td>
</tr>
<tr>
<td>MAX(A1, A2, A3, ...)</td>
<td>547</td>
</tr>
<tr>
<td>MAXEXPONENT(X)</td>
<td>548</td>
</tr>
<tr>
<td>MAXLOC(ARRAY, DIM, MASK)</td>
<td>549</td>
</tr>
<tr>
<td>MINLOC(ARRAY, DIM, MASK)</td>
<td>549</td>
</tr>
<tr>
<td>MAXVAL(ARRAY, DIM, MASK)</td>
<td>550</td>
</tr>
<tr>
<td>MINVAL(ARRAY, DIM, MASK)</td>
<td>550</td>
</tr>
<tr>
<td>MERGE(STRING, STRING, STRING)</td>
<td>552</td>
</tr>
<tr>
<td>MIN(A1, A2, A3, ...)</td>
<td>552</td>
</tr>
<tr>
<td>MINEXponent(X)</td>
<td>553</td>
</tr>
<tr>
<td>MINLOC(ARRAY, DIM, MASK)</td>
<td>553</td>
</tr>
<tr>
<td>MINLOC(ARRAY, MASK)</td>
<td>554</td>
</tr>
<tr>
<td>MINVAL(ARRAY, DIM, MASK)</td>
<td>556</td>
</tr>
<tr>
<td>MOD(A, P)</td>
<td>557</td>
</tr>
<tr>
<td>MODULO(A, P)</td>
<td>558</td>
</tr>
<tr>
<td>MOVBITs(FROM, FROMPOS, LEN, TO, TOPoS)</td>
<td>558</td>
</tr>
<tr>
<td>NEAREST(X)</td>
<td>559</td>
</tr>
<tr>
<td>NEW_LINE(A)</td>
<td>560</td>
</tr>
<tr>
<td>NINT(A, KIND)</td>
<td>561</td>
</tr>
<tr>
<td>NOT(I)</td>
<td>561</td>
</tr>
<tr>
<td>NULL.MOLD)</td>
<td>562</td>
</tr>
<tr>
<td>NUM.PARTHDS()</td>
<td>563</td>
</tr>
<tr>
<td>NUMBER_OF_PROCESSORS</td>
<td>564</td>
</tr>
<tr>
<td>NUM_USRTHDS()</td>
<td>565</td>
</tr>
<tr>
<td>PACK(ARRAY, MASK, VECTOR)</td>
<td>565</td>
</tr>
<tr>
<td>POPCNT(I)</td>
<td>566</td>
</tr>
<tr>
<td>POPPAR(I)</td>
<td>567</td>
</tr>
<tr>
<td>PRECISION(X)</td>
<td>568</td>
</tr>
<tr>
<td>PRESENT(A)</td>
<td>569</td>
</tr>
<tr>
<td>PROCESSORS_SHAPE()</td>
<td>569</td>
</tr>
<tr>
<td>PRODUCT(ARRAY, DIM, MASK)</td>
<td>570</td>
</tr>
<tr>
<td>QCmplX(X, Y)</td>
<td>572</td>
</tr>
<tr>
<td>QEXT(A)</td>
<td>572</td>
</tr>
<tr>
<td>RADIX(X)</td>
<td>573</td>
</tr>
<tr>
<td>RAND()</td>
<td>574</td>
</tr>
<tr>
<td>RANDOM_NUMBER</td>
<td>574</td>
</tr>
<tr>
<td>RANDOM_SEED(SIZE, PUT, GET, GENERATOR)</td>
<td>575</td>
</tr>
<tr>
<td>RANGE(X)</td>
<td>576</td>
</tr>
<tr>
<td>REAL(A, KIND)</td>
<td>577</td>
</tr>
<tr>
<td>REPEAT(STRING, NCopies)</td>
<td>578</td>
</tr>
<tr>
<td>RESHAPE</td>
<td>579</td>
</tr>
<tr>
<td>RRSPPACING(X)</td>
<td>580</td>
</tr>
<tr>
<td>RSHIFT(L, SHIFT)</td>
<td>580</td>
</tr>
<tr>
<td>SCALE(X)</td>
<td>581</td>
</tr>
<tr>
<td>SCAN(STRING, SET, BACK)</td>
<td>582</td>
</tr>
<tr>
<td>SELECTED_INT_KIND</td>
<td>582</td>
</tr>
<tr>
<td>SELECTED_REAL_KIND</td>
<td>583</td>
</tr>
<tr>
<td>SET_EXPONENT</td>
<td>584</td>
</tr>
<tr>
<td>SHAPE</td>
<td>585</td>
</tr>
<tr>
<td>SIGN(A, B)</td>
<td>585</td>
</tr>
<tr>
<td>SIGNAL(I, PROC)</td>
<td>587</td>
</tr>
<tr>
<td>SIN(X)</td>
<td>587</td>
</tr>
<tr>
<td>SIND(X)</td>
<td>588</td>
</tr>
<tr>
<td>SINH(X)</td>
<td>589</td>
</tr>
<tr>
<td>SIZE(ARRAY, DIM)</td>
<td>590</td>
</tr>
<tr>
<td>SIZEOF(A)</td>
<td>590</td>
</tr>
<tr>
<td>SPACING(X)</td>
<td>592</td>
</tr>
<tr>
<td>SPREAD(ARRAY, DIM, NCopies)</td>
<td>592</td>
</tr>
<tr>
<td>SQRT(X)</td>
<td>593</td>
</tr>
<tr>
<td>SRAND(SEED)</td>
<td>594</td>
</tr>
<tr>
<td>SUM(ARRAY, DIM, MASK)</td>
<td>595</td>
</tr>
<tr>
<td>SYSTEM(CMD, RESULT)</td>
<td>596</td>
</tr>
<tr>
<td>SYSTEM_CLOCK(COUNT, COUNT_RATE, COUNT_MAX)</td>
<td>597</td>
</tr>
<tr>
<td>TAN(X)</td>
<td>598</td>
</tr>
<tr>
<td>TAND(X)</td>
<td>599</td>
</tr>
<tr>
<td>TANH(X)</td>
<td>600</td>
</tr>
<tr>
<td>TINY(X)</td>
<td>600</td>
</tr>
<tr>
<td>TRANSFER(SOURCE, MOLD, SIZE)</td>
<td>601</td>
</tr>
<tr>
<td>TRNSPOSE(MATRIX)</td>
<td>602</td>
</tr>
<tr>
<td>TRM(STRING)</td>
<td>603</td>
</tr>
<tr>
<td>UBOUND(ARRAY, DIM)</td>
<td>603</td>
</tr>
<tr>
<td>UNBOUND(ARRAY, DIM)</td>
<td>604</td>
</tr>
<tr>
<td>VERIFY(STRING, SET, BACK)</td>
<td>605</td>
</tr>
</tbody>
</table>

**Chapter 14. Hardware-specific intrinsic procedures**
Chapter 15. VMX intrinsic procedures

VEC_ABS(ARG1) ........................................ 623
VEC_ABS2(ARG1) ...................................... 623
VEC_ADD(ARG1, ARG2) .................................. 624
VEC_ADDC(ARG1, ARG2) ................................ 624
VEC_ALL_EQ(ARG1, ARG2) ................................ 625
VEC_ALL_GE(ARG1, ARG2) ................................ 626
VEC_ALL_GT(ARG1, ARG2) ................................. 626
VEC_ALL_IN(ARG1, ARG2) ................................ 627
VEC_ALL_LE(ARG1, ARG2) ................................ 627
VEC_ALL_LT(ARG1, ARG2) ................................ 628
VEC_ALL_NAN(ARG1) ..................................... 628
VEC_ALL_NE ARG1, ARG2) ................................ 628
VEC_ALL_NGE(ARG1, ARG2) ................................ 629
VEC_ALL_NGT(ARG1, ARG2) ................................ 629
VEC_ALL_NLE(ARG1, ARG2) ................................ 630
VEC_ALL_NLT(ARG1, ARG2) ................................ 630
VEC_ALL_NUMERIC(ARG1) ................................ 631
VEC_AND2(ARG1, ARG2) .................................. 631
VEC_ANDC(ARG1, ARG2) .................................. 632
VEC_ANY_EQ(ARG1, ARG2) ................................ 632
VEC_ANY_GE(ARG1, ARG2) ................................ 632
VEC_ANY_GT(ARG1, ARG2) ................................ 633
VEC_ANY_LE(ARG1, ARG2) ................................ 634
VEC_ANY_LT(ARG1, ARG2) ................................ 634
VEC_ANY_NAN(ARG1) ..................................... 634
VEC_ANY_NE(ARG1, ARG2) ................................ 635
VEC_ANY_NGE(ARG1, ARG2) ................................ 635
VEC_ANY_NGT(ARG1, ARG2) ................................ 636
VEC_ANY_NLE(ARG1, ARG2) ................................ 636
VEC_ANY_NLT(ARG1, ARG2) ................................ 637
VEC_ANY_NUMERIC(ARG1) ................................ 637
VEC_ANY_OUT(ARG1, ARG2) ................................ 637
VEC_AVG(ARG1, ARG2) .................................... 638
VEC_CEIL(ARG1) .......................................... 638
VEC_CMPB(ARG1, ARG2) .................................. 639
VEC_CMPEQ(ARG1, ARG2) .................................. 640
VEC_CMPGE(ARG1, ARG2) .................................. 640
VEC_CMPGT(ARG1, ARG2) .................................. 641
VEC_CMPLE(ARG1, ARG2) .................................. 641
VEC_CMPLT(ARG1, ARG2) .................................. 642
VEC_CONVERT(V, MOLD) .................................... 642
VEC_CTRF(ARG1, ARG2) ................................... 643
VEC_CTS(ARG1, ARG2) ..................................... 643
VEC_CTX(ARG1, ARG2) ..................................... 644
VEC_DSS(ARG1) ............................................ 644
VEC_DSSALL .............................................. 644
VEC_DST(ARG1, ARG2, ARG3) ............................ 645
VEC_DSTARG1, ARG2, ARG3) .............................. 645
VEC_DSTT(ARG1, ARG2, ARG3) ............................ 645
VEC_DSTTARG1, ARG2, ARG3) ............................. 646
VEC_EXPTE(ARG1) ........................................ 646
VEC_FLOOR(ARG1) ....................................... 647
VEC_IDL(ARG1, ARG2) ..................................... 647
VEC_IDE(ARG1, ARG2) ..................................... 648
VEC_IDLG(ARG1, ARG2) .................................... 648
VEC_LOG2(ARG1) .......................................... 649
VEC_LVSL(ARG1, ARG2) .................................... 649
VEC_LVSR(ARG1, ARG2) .................................... 650
VEC_MADDARG1, ARG2, ARG3) ............................ 650
VEC_MADD2(ARG1, ARG2) .................................. 651
VEC_MAX(ARG1, ARG2) ..................................... 651
VEC_MERGEH(ARG1, ARG2) ................................ 652
VEC_MERGEL(ARG1, ARG2) ................................. 652
VEC_MFVSCR .............................................. 653
VEC_MIN(ARG1, ARG2) ..................................... 653
VEC_MLADD(ARG1, ARG2, ARG3) .......................... 654
VEC_MRAADD3(ARG1, ARG2, ARG3) ....................... 654
VEC_MSUM(ARG1, ARG2, ARG3) ......................... 655
VEC_MSUM2(ARG1, ARG2, ARG3) ......................... 655
VEC_MTVSCR(ARG1) ....................................... 656
VEC_MUL2(ARG1, ARG2) ................................... 656
VEC_MUL22(ARG1, ARG2) .................................. 657
VEC_NMSUB(ARG1, ARG2, ARG3) ......................... 657
VEC_NOR(ARG1, ARG2) .................................... 658
VEC_OR(ARG1, ARG2) ...................................... 658
VEC_PACK(ARG1, ARG2) .................................... 659
VEC_PACK2(ARG1, ARG2) ................................... 659
VEC_PACKS(ARG1, ARG2) ................................... 660
VEC_PACKSU(ARG1, ARG2) ................................ 660
VEC_PERM(ARG1, ARG2, ARG3) ............................ 661
VEC.RE(ARG1) ........................................... 661
VEC.RL(ARG1, ARG2) ..................................... 662
VEC_ROUND(ARG1) ........................................ 662
VEC_RSQRT2(ARG1) ....................................... 663
VEC_SEL(ARG1, ARG2, ARG3) ............................. 663
VEC_SL(ARG1, ARG2) ...................................... 664
VEC_SLD(ARG1, ARG2, ARG3) ............................. 664
VEC_SLL(ARG1, ARG2) ..................................... 665
VEC_SLO(ARG1, ARG2) ..................................... 665
VEC_SPLAT2(ARG1, ARG2) ................................ 666
VEC_SPLAT2S8(ARG1) ..................................... 666
VEC_SPLAT32(ARG1) ....................................... 667
VEC_SPLATU8(ARG1) ....................................... 667
Chapter 16. Language interoperability
features ........................................ 679
Interoperability of types ........................................ 679
   Intrinsic types ........................................ 679
   Derived types ........................................ 679
Interoperability of Variables .................................... 680
Interoperability of common blocks ................................ 680
Interoperability of procedures .................................. 681
The ISO_C_BINDING module .................................... 681
   Constants for use as kind type parameters ............... 681
   Character constants .................................... 683
   Other constants ....................................... 683
   Types ................................................. 683
   Procedures ........................................... 683
Binding labels ................................................. 685

Chapter 17. The ISO_FORTRAN_ENV
intrinsic module ............................................. 687
CHARACTER_STORAGE_SIZE ................................. 687
ERROR_UNIT .............................................. 687
FILE_STORAGE_SIZE ....................................... 687
INPUT_UNIT .............................................. 688
IOSTAT_END ............................................... 688
IOSTAT_EOR ............................................... 688
NUMERIC_STORAGE_SIZE .................................... 689
OUTPUT_UNIT .............................................. 689

Chapter 18. Floating-point control and
inquiry procedures ......................................... 691
fpsets .................................................... 691
Efficient floating-point control and inquiry
procedures ............................................... 692
xf_fp_util Floating-Point Procedures ....................... 693
IEEE Modules and support ................................... 696
   Compiling and exception handling ....................... 696
   General rules for implementing IEEE modules ......... 697
IEEE Derived data types and constants ..................... 697
IEEE Operators .......................................... 699
IEEE PROCEDURES ...................................... 699
Rules for floating-point status .............................. 715
Examples .................................................. 716

Chapter 19. Service and utility
procedures ............................................... 721
General service and utility procedures ....................... 721
List of service and utility procedures ....................... 722
alarm (time, func) ....................................... 722
bic (X1, X2) ............................................ 723
bis (X1, X2) ............................................. 723
bit (X1, X2) ............................................. 723
clock () .................................................. 724
ctime (STR, TIME) ....................................... 724
date () ................................................... 724
dtime (dtime_struct) ..................................... 725
etime (etime_struct) ...................................... 725
exit (exit_status) ........................................ 726
ftime (str) ............................................... 726
fiosetup (unit, command, argument) ......................... 726
flush (lunit) ............................................. 727
fseek (lunit) .............................................. 728
ftime64 (lunit) .......................................... 728
getarg (i1, c1) ........................................... 729
getcwd (name) ............................................ 730
gmtime (st ime, timer) ................................... 730
gmtime (time, tarray) ................................... 731
hostnm (name) ............................................ 732
iargc () .................................................. 733
idate (idate struct) ..................................... 733
ierrno () .................................................. 733
irand () ................................................. 733
irc () ..................................................... 734
itime (itime struct) ...................................... 734
jdate () ................................................... 735
lendr _ (str) ............................................. 735
lnlink (_str) ............................................ 735
ltime _ (st ime, timer) .................................. 736
mtime () .................................................. 736
qsort _ (array, len, isize, compar) ......................... 737
qsort _ down (array, len, isize) .......................... 737
qsort _ up (array, len, isize) ............................ 738
rtc () .................................................... 739
setreopts (c1) ........................................... 739
sleep (_sec) ............................................. 739
time () ................................................... 739
timef () .................................................. 740
timef _ delta (t) ........................................ 740
umask (_cmask) .......................................... 741
usleep (_msec) .......................................... 741
xl__trbk () ............................................. 741
Appendix A. Compatibility across
standards ................................................. 743
Fortran 90 compatibility .................................... 744
Obsolescent features ...................................... 744
Deleted features ........................................ 746
About this document

This document, which is part of the XL Fortran documentation suite, describes the syntax, semantics, and IBM® implementation of the Fortran programming language on the Linux® operating system. Although XL Fortran implementations conform to Fortran 95, parts of Fortran 2003, and other specifications maintained by the ISO standards for the Fortran programming language, they also incorporate many extensions to the core language. These extensions have been implemented with the aims of enhancing usability in specific operating environments, assuring compatibility with other compilers, and supporting new hardware capabilities.

Who should read this document

This document is a reference for users who already have experience programming in Fortran. Users new to Fortran can still use this document to find information on the language and features unique to XL Fortran; however, it does not aim to teach programming concepts nor to promote specific programming practices.

How to use this document

While this document covers both standard and implementation-specific features of XL Fortran, it does not include information on the following topics, which are covered in other documents:

- Installation, system requirements, last-minute updates: see the [XL Fortran Advanced Edition V10.1 for Linux Installation Guide](https://www.ibm.com) and product README.
- Overview of XL Fortran features: see the [Getting Started with XL Fortran Advanced Edition V10.1 for Linux](https://www.ibm.com).
- Compiler setup, compiling / running programs, compiler options, diagnostics: see the [XL Fortran Compiler Reference](https://www.ibm.com).
- Optimizing, porting, OpenMP / SMP programming: see the [XL Fortran Optimization and Programming Guide](https://www.ibm.com).
- Operating system commands related to the use of the compiler: consult your Linux-specific distribution’s man page help and documentation.

How this document is organized

The following lists group information into sections that provide detail on particular language topics and implementations:

- XL Fortran language elements:
  - Fundamentals of the XL Fortran language
  - Data types and objects
  - Arrays
  - Expressions and assignment
  - Execution control
  - Program units and procedures
  - Understanding XL Fortran Input/Output
  - Input/Output formatting
  - Statements and attributes
  - General directives
Intrinsic procedures

Language interoperability features

- Procedures that provide hardware-related functionality, and additional features for those already familiar with the Fortran language:
  - Floating-point control and inquiry procedures
  - Hardware–specific directives
  - Hardware–specific intrinsic procedures
  - The ISO_FORTRAN_ENV intrinsic module
  - Service and utility procedures

- The appendices provide information on compatibility across standards for users of earlier versions of Fortran, and the ASCII and EBCDIC character sets mapping table.

Conventions and terminology used in this document

Typographical conventions

The following table explains the typographical conventions used in this document.

Table 1. Typographical conventions

<table>
<thead>
<tr>
<th>Typeface</th>
<th>Indicates</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>bold</strong></td>
<td>Commands, executable names, and compiler options.</td>
<td>By default, if you use the <code>-qsmp</code> compiler option in conjunction with one of these invocation commands, the option <code>-qdirective=IBM*:SMP$:$OMP:IBMP:IBMT</code> will be on.</td>
</tr>
<tr>
<td><em>italics</em></td>
<td>Parameters or variables whose actual names or values are to be supplied by the user. Italics are also used to introduce new terms.</td>
<td>The maximum length of the <code>trigger_constant</code> in fixed source form is 4 for directives that are continued on one or more lines.</td>
</tr>
<tr>
<td><strong>UPPERCASE bold</strong></td>
<td>Fortran programming keywords, statements, directives, and intrinsic procedures.</td>
<td>The <code>ASSERT</code> directive applies only to the <code>DO</code> loop immediately following the directive, and not to any nested <code>DO</code> loops.</td>
</tr>
<tr>
<td><strong>lowercase bold</strong></td>
<td>Lowercase programming keywords and library functions, compiler intrinsic procedures, file and directory names, examples of program code, command strings, or user-defined names.</td>
<td>If you call <code>omp_destroy_lock</code> with an uninitialized lock variable, the result of the call is undefined.</td>
</tr>
</tbody>
</table>

Qualifying elements (icons and bracket separators)

This document uses icons to delineate small segments of text as follows:
### Table 2. Qualifying elements

<table>
<thead>
<tr>
<th>Icon</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>![F2003]</td>
<td>The text describes an IBM XL Fortran implementation of the Fortran 2003 standard.</td>
</tr>
<tr>
<td>![F95]</td>
<td>The text describes an IBM XL Fortran implementation of the Fortran 95 standard.</td>
</tr>
<tr>
<td>![IBM]</td>
<td>The text describes a feature that is an IBM XL Fortran compiler extension to the standard language specifications.</td>
</tr>
</tbody>
</table>

This document uses marked bracket separators to delineate large blocks of standards-specific and extensions text as follows:

#### Fortran 2003 Standard

Text delineating Fortran 2003 standard-specific information.

---

End of Fortran 2003 Standard

#### Fortran 95

Text delineating Fortran 95 standard-specific information.

---

End of Fortran 95

#### IBM Extension

Text delineating extensions to the FORTRAN 77, Fortran 90, Fortran 95 and Fortran 2003 standards, where an extension is any processor dependent value or behavior.

---

End of IBM Extension

### How to read syntax diagrams

Throughout this document, diagrams illustrate XL Fortran syntax. This section will help you to interpret and use those diagrams.

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

You must enter punctuation marks, parentheses, arithmetic operators, and other special characters as part of the syntax.

- Read syntax diagrams 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 continues on the next line.
  - The ▶️ symbol indicates that a statement continues 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.
- IBM XL Fortran extensions to and implementations of language standards are marked by a number in the syntax diagram with an explanatory note immediately following the diagram.

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

  ![Required items diagram]

- Optional items are shown below the main path:

  ![Optional items diagram]

**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 are shown vertically, in a stack. If you *must* choose one of the items, one item of the stack is shown on the main path:

  ![Choice stack diagram]

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

  ![Optional choice stack diagram]

- An arrow returning to the left above the main line (a repeat arrow) indicates that you can repeat an item, 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 items in the stack.

Sample syntax diagram
The following is an example of a syntax diagram with an interpretation:

Interpret the diagram as follows:
• Enter the keyword EXAMPLE.
• EXAMPLE is an IBM extension.
• 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.)

Examples
The examples in this document are coded in a simple style that does not try to conserve storage, check for errors, achieve fast performance, or demonstrate recommended practice.
Related information

**IBM XL Fortran documentation**

XL Fortran provides product documentation in the following formats:

- **Readme files**
  Readme files contain late-breaking information, including changes and corrections to the product documentation. Readme files are located by default in the /opt/ibmcmp/xlf/10.1 directory and in the root directory of the installation CD.

- **Installable man pages**
  Man pages are provided for the compiler invocations and all command-line utilities provided with the product. Instructions for installing and accessing the man pages are provided in the XL Fortran Advanced Edition V10.1 for Linux Installation Guide.

- **Information center**
  The information center of searchable HTML files can be launched on a network and accessed remotely or locally. Instructions for installing and accessing the information center are provided in the XL Fortran Advanced Edition V10.1 for Linux Installation Guide. The information center is also viewable on the Web at: http://publib.boulder.ibm.com/infocenter/lnxpcomp/index.jsp

- **PDF documents**
  PDF documents are located by default in the /opt/ibmcmp/xlf/10.1/doc/language/pdf directory, where language can be one of the following supported languages:
  - en_US (U.S. English)
  - ja_JP (Japanese)
  PDF documents are also available on the Web at: http://www.ibm.com/software/awdtools/fortran/xlfortran/library

In addition to this document, the following files comprise the full set of XL Fortran product manuals:

<table>
<thead>
<tr>
<th>Table 3. XL Fortran PDF files</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Document title</strong></td>
</tr>
<tr>
<td>IBM XL Fortran Advanced Edition V10.1 for Linux Installation Guide</td>
</tr>
<tr>
<td>Getting Started with IBM XL Fortran Advanced Edition V10.1 for Linux</td>
</tr>
<tr>
<td>IBM XL Fortran Advanced Edition V10.1 for Linux Compiler Reference</td>
</tr>
</tbody>
</table>
Table 3. XL Fortran PDF files (continued)

<table>
<thead>
<tr>
<th>Document title</th>
<th>PDF file name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM XL Fortran Advanced Edition V10.1 for Linux</td>
<td>opg.pdf</td>
<td>Contains information on application optimization and tuning, advanced programming topics, such as application porting, interlanguage calls, floating-point operations, input/output, and parallelization, and the XL Fortran high-performance libraries.</td>
</tr>
</tbody>
</table>

These PDF files are viewable and printable from Adobe Reader. If you do not have the Adobe Reader installed, you can download it from [http://www.adobe.com](http://www.adobe.com).

**Additional documentation**

More documentation related to XL Fortran, including redbooks, whitepapers, tutorials, and other articles is available on the Web at:


Operating system and other documentation is available as follows:

- The RPM [home page](http://www.rpm.org/) covers all aspects of the standard Linux installation procedure using the RPM Package Manager (RPM).
- For general information and documentation on Linux, visit [The Linux Documentation Project](http://www.tldp.org/) Consult your Linux-specific distribution’s man page help and documentation for information on using your operating system and its features.
- System V Application Binary Interface: PowerPC Processor Supplement is a supplement to the generic System V ABI and contains information specific to System V implementations built on the PowerPC Architecture™ operating in 32-bit mode.
- 64-bit PowerPC ELF Application Binary Interface Supplement is a supplement to the generic System V ABI and contains information specific to System V implementations built on the PowerPC Architecture operating in 64-bit mode.

**Related documentation**

You can also consult the following publications, which are also referenced throughout this document:

- ESSL for Linux on POWER V4.2 Guide and Reference

**Standards documents**

XL Fortran is designed according to the following standards. You can refer to these standards for precise definitions of some of the features found in this document.

- Federal (USA) Information Processing Standards Publication Fortran, FIPS PUB 69-1.
- Information technology - Programming languages - Fortran - Part 1: Base language, ISO/IEC 1539-1:1997. (This document uses its informal name, Fortran 95.)
• Military Standard Fortran DOD Supplement to ANSI X3.9-1978, MIL-STD-1753 (United States of America, Department of Defense standard). Note that XL Fortran supports only those extensions documented in this standard that have also been subsequently incorporated into the Fortran 90 standard.
• OpenMP Application Program Interface, Version 2.5 (May 2005) specification.

Technical support

Additional technical support is available from the XL Fortran Support page. This page provides a portal with search capabilities to a large selection of technical support FAQs and other support documents. You can find the XL Fortran Support page on the Web at:

http://www.ibm.com/software/awdtools/fortran/xlfortran/support

If you cannot find what you need, you can e-mail:

compinfo@ca.ibm.com

For the latest information about XL Fortran, visit the product information site at:

http://www.ibm.com/software/awdtools/fortran/xlfortran

How to send your comments

Your feedback is important in helping to provide accurate and high-quality information. If you have any comments about this document or any other XL Fortran documentation, send your comments by e-mail to:

compinfo@ca.ibm.com.

Be sure to include the name of the document, the part number of the document, the version of XL Fortran, and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).
Chapter 1. XL Fortran for Linux

The Language Reference is part of a documentation suite that offers information on installing and using the IBM XL Fortran compiler on Linux. This document defines the syntax, semantics, and restrictions you must follow to write valid XL Fortran programs.

Fortran (FORmula TRANslation) is a high-level programming language primarily useful for engineering, mathematical, and scientific applications involving numeric computations.

XL Fortran implements many Fortran 95, Fortran 2003, and other language specifications maintained by the ISO standards for the Fortran programming language, and also incorporates many extensions to the core language. These extensions have been implemented with the aims of enhancing usability in specific operating environments, assuring compatibility with other compilers, and supporting new hardware capabilities. In theory, a program that compiles correctly on one standards-conforming compiler will compile and execute correctly under all other conforming compilers, insofar as hardware differences permit.

The compiler detects most non-conformities to the XL Fortran language rules, but may not detect some syntactic and semantic combinations. The compiler cannot detect all combinations for 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 the programs run as expected.

Language standards

This section briefly summarizes the XL Fortran language standard implementations it follows. The Qualifying elements section contains details on how XL Fortran marks language standard specific information.

Fortran 2003 Standard

Segments of this document contain information based on the Fortran 2003 Standard. The standard is open to continual interpretation, modification and revision. IBM reserves the right to modify the behavior of any features of this product to conform with future interpretations of this standard.

Fortran 95

The Fortran 95 language standard is upward-compatible with the FORTRAN 77 and Fortran 90 language standards, excluding deleted features. Some of the improvements provided by the Fortran 95 standard are:

- Default initialization
- ELEMENTAL procedures.
- The FORALL construct statement.
- POINTER initialization.
- PURE functions.
- Specification expressions
The Fortran standard committees respond to questions of interpretation about aspects of Fortran. Some questions can relate to language features already implemented in the XL Fortran compiler. Any answers given by these committees relating to these language features can result in changes to future releases of the XL Fortran compiler, even if these changes result in incompatibilities with previous releases of the product.

**Fortran 90**

Fortran 90 offers many new features and feature enhancements to FORTRAN 77. The following topics outline some of the key features that Fortran 90 brings to the FORTRAN 77 language:

- Array enhancements
- Control construct enhancements
- Derived types
- Dynamic behavior
- Free source form
- Modules
- Parameterized data types
- Procedure enhancements
- Pointers

**IBM Extensions**

An IBM extension generally modifies a rule or restriction from a given standards implementation. In this document, IBM extensions to the Fortran 2003, Fortran 90, and Fortran 95 standards are marked as indicated in the Qualifying elements section.

**Other standards and standards documents**

**OpenMP API Version 2.5**

The OpenMP API provides additional features which you can use to supplement the existing FORTRAN 77, Fortran 90 and Fortran 95 language standards.

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

You can find information pertaining to the implementation of OpenMP API Version 2.5 in the following sections:

- OpenMP environment variables in the XL Fortran Optimization and Programming Guide
- SMP Directives in the XL Fortran Optimization and Programming Guide

**Standards documents**

XL Fortran is designed according to the standards listed in the Standards documents section. You can refer to these standards for precise definitions of some of the features found in this document.
Chapter 2. Fundamentals of the XL Fortran language

This section describes the fundamentals of an XL Fortran program:

- “Characters”
- “Names” on page 4
- “Statements” on page 5
- “Lines and source formats” on page 5
- “Order of statements and execution sequence” on page 14

Characters

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

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

Note: * Lower case letters are used in XL Fortran

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 747 for a complete listing of the ASCII character set.)

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 5 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.

IBM Extension

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 Linux 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 1a iA IA

the same by default, but treats them as distinct identifiers if you specify the -qmixed compiler option.

End of IBM Extension

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 72)
- An array section (see “Array sections” on page 73)
- A structure component (see “Structure components” on page 34)
- A character substring (see “Character substrings” on page 27)
Statements

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

IBM Extension

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

End of IBM Extension

See Chapter 10, “Statements and attributes,” on page 235 for more information on 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.

IBM Extension

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

End of IBM Extension

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 129 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.

IBM Extension

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
The kinds of lines are:

<table>
<thead>
<tr>
<th>Line Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial Line</td>
<td>Is the first line of a statement.</td>
</tr>
<tr>
<td>Continuation Line</td>
<td>Continues a statement beyond its initial line.</td>
</tr>
<tr>
<td>Comment Line</td>
<td>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.</td>
</tr>
<tr>
<td>* Conditional Compilation Line</td>
<td>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 11)</td>
</tr>
<tr>
<td>* Debug Line</td>
<td>Indicates that the line is for debugging code (for fixed source form only). In XL Fortran the letter D or X must be specified in column 1. (See “Debug lines” on page 8)</td>
</tr>
<tr>
<td>* Directive Line</td>
<td>Provides instructions or information to the compiler in XL Fortran. (See “Comment form directives” on page 433)</td>
</tr>
</tbody>
</table>

In XL Fortran, 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`, `fort77`, `xlf`, `xlf_r` invocation commands.

Fortran 90 free source form is the default when using the `xlf90`, `xlf90_r`, `xlf95`, or `xlf95_r` invocation commands.

See `Compiling XL Fortran Programs` in the `XL Fortran Compiler Reference` for details on invocation commands.

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 in XL Fortran by using the `qfixed=right_margin` compiler option (see the `XL Fortran Compiler`
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 in XL Fortran, 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, a D or an X 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 in XL Fortran:
  – Columns 1 through 6 begin with either blanks, a statement label, a D or an X 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) in XL Fortran. Otherwise,
If Example specify debugging initial line options:

1. Debugging

A Debug statements

Statements

Separator

only semicolon

in A you debug

semicolon

blank,

– Column 6 must have either a nonzero character or a nonwhite space character. 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.

IBM Extension

– In XL Fortran 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

IBM Extension

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

• 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 specify -qxlines, the compiler interprets the X in column 1 as a blank and treats these lines as source code.

• If you do not specify -qdlines or -qxlines, 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 continuation character as well as a D or an X in column 1. 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 or -qxlines compiler option.

Example of fixed source form:
Free source form

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

IBM Extension

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

End of IBM Extension

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 non-comment line that follows the initial 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.
v 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. The following table lists
keywords that require white space, and keywords for which white space is
optional.
Table 5. Keywords where white space is optional
BLOCK DATA
DOUBLE COMPLEX
DOUBLE
PRECISION
ELSE IF
ELSE WHERE
END ASSOCIATE
END BLOCK DATA
END DO

END ENUM
END FILE
END FORALL

END PROGRAM
END SELECT
END STRUCTURE

END
END
END
END
END

END SUBROUTINE
END TYPE
END UNION
END WHERE
GO TO

FUNCTION
IF
INTERFACE
MAP
MODULE

IN OUT
SELECT CASE

See “Type Declaration” on page 409 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

10

XL Fortran Language Reference

! 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
"23456789012345678901234567890123456789012345678901234567890123456789010
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
```

End of IBM Extension

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:

- S, C$, c$, or *S, for fixed source form; or
- S, 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 -directive='S' option, then lines that start with the trigger, such as S$, 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 6 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 9 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.
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.

```fortran
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
!
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
```

In the "Editing, Compiling, Linking, and Running XL Fortran Programs" section of the [XL Fortran Compiler Reference](#) for an example of using this macro.
## Order of statements and execution sequence

**Table 6. Statement order**

<table>
<thead>
<tr>
<th>1</th>
<th>PROGRAM, FUNCTION, SUBROUTINE, MODULE, or BLOCK DATA Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>USE Statements</td>
</tr>
<tr>
<td>3</td>
<td>IMPORT Statements</td>
</tr>
<tr>
<td>4</td>
<td>DATA, FORMAT, and ENTRY Statements</td>
</tr>
<tr>
<td>5</td>
<td>Derived-Type Definitions, Interface Blocks, Type Declaration Statements, Specification Statements, IMPLICIT Statements, and PARAMETER Statements</td>
</tr>
<tr>
<td>6</td>
<td>Executable constructs</td>
</tr>
<tr>
<td>7</td>
<td>CONTAINS Statement</td>
</tr>
<tr>
<td>8</td>
<td>Internal Subprograms or Module Subprograms</td>
</tr>
<tr>
<td>9</td>
<td>END Statement</td>
</tr>
</tbody>
</table>

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 document 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 document.

Refer to [Chapter 7, “Program units and procedures,” on page 129](#) or [Chapter 10, “Statements and attributes,” on page 235](#) 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 document, 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 section describes:
- “Data types”
- “Data objects”
- “Intrinsic types” on page 16
- “Derived types” on page 28
- “Typeless literal constants” on page 46
- “How type is determined” on page 51
- “Definition status of variables” on page 52
- “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

A derived type is a user-defined data type. The components of a derived type can be a mixture of both intrinsic and 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*.

The KIND intrinsic function returns the kind type parameter of its argument. See “KIND(X)” on page 535 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 and ENUM). 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

IBM Extension

The following table shows the range of values that XL Fortran can represent using the integer data type:
<table>
<thead>
<tr>
<th>Kind parameter</th>
<th>Range of values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-128 through 127</td>
</tr>
<tr>
<td>2</td>
<td>-32,768 through 32,767</td>
</tr>
<tr>
<td>4</td>
<td>-2,147,483 through 2,147,483</td>
</tr>
<tr>
<td>8</td>
<td>-9,223,372,036 through 9,223,372,036</td>
</tr>
</tbody>
</table>

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.

--- End of IBM Extension ---

The integer type specifier must include the `INTEGER` keyword. See "INTEGER" on page 336 for details on declaring entities of type integer.

The form of a signed integer literal constant is:

```
[s] _digit_ _kind_param_
```

`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`.

--- IBM Extension ---

If no `kind_param` is specified in XL Fortran, 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.

--- End of IBM Extension ---

**Examples of integer constants**

0  ! has default integer size
-173 2  ! 2-byte constant
922372036054775807  ! Kind type parameter is promoted to 8
The following table shows the range of values that XL Fortran can represent with the real data type:

<table>
<thead>
<tr>
<th>Kind Parameter</th>
<th>Approximate Absolute Nonzero Minimum</th>
<th>Approximate Absolute Maximum</th>
<th>Approximate Precision (decimal digits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1.175494E-38</td>
<td>3.402823E+38</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>2.225074D-308</td>
<td>1.797693D+308</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>2.225074Q-308</td>
<td>1.797693Q+308</td>
<td>31</td>
</tr>
</tbody>
</table>

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.

For REAL(8) numbers, the HUGE intrinsic returns 0x7FEFFFFFFFFFFFF and the TINY intrinsic returns 0x0010000000000000. As a result, we have INT(MIN(308.254715559916747, 307.652655568588784)), and therefore the range is 307. Note that the LOG scale is not symmetric on both ends of the exponent.

The IBM format of REAL(16) numbers is composed of two REAL(8) numbers of different magnitudes that do not overlap. That is, the binary exponents differ by at least the number of fraction bits in a REAL(8).

For REAL(16), the RANGE intrinsic returns the range of the numbers that have both REAL(8) numbers normalized. Consequently, for REAL(16) numbers, the HUGE intrinsic returns 0x7FEFFFFFFFFFFFF7C9FFFFFFFF and the TINY intrinsic returns 0x00360000000000000000000000000000. As a result, we have INT(MIN(308.25471555991674389886862819788120, 291.69806579839777816211298898803388)), where the range is 291.

308 is the lowest or highest exponent that can be represented in the REAL(8) or REAL(16) numbers.
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 or a signaling NaN. See [Implementation details of XL Fortran floating-point processing](#) in the [XL Fortran Optimization and Programming Guide](#) for details on the internal representation of NaN values.

---

End of IBM Extension

---

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 385 and “DOUBLE PRECISION” on page 286 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:

```
  + - digit exponent
  + - digit . exponent
  + - digit . digit exponent
  + - digit exponent
  - - kind_param
```

exponent
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) in XL Fortran.

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
Example 1:
+0.

Example 2:
+5.432E02_16 !543.2 in 16-byte representation

Example 3:
7.E3

--- IBM Extension ---

Example 4:
3.40-301
! Extended-precision constant

--- End of IBM Extension ---

Vector

--- IBM Extension ---

An entity you declare using the VECTOR keyword as part of a type declaration statement is of a vector type. An entity of a vector type has the same type as another entity if both entities are vectors that contain elements of the same type and kind. Otherwise, the two entities are of different types. You must not include vector data types in a NAMELIST statement, or formatted directed I/O formatting.

A vector type can be any of the following:

- A PIXEL vector.
- An INTEGER vector with a kind type parameter of 1, 2 or 4.
- An UNSIGNED vector with a kind type parameter of 1, 2, or 4.
- A REAL vector with a kind parameter of 4.
A vector must only be declared if your -qarch setting specifies a VMX-enabled architecture and you compile with -qenablevmx.

A vector is always a 16-byte entity. Consequently, an INTEGER(1) or UNSIGNED(1) vector contains 16 elements. INTEGER(2), UNSIGNED(2), or PIXEL, vectors contain 8 elements, while INTEGER(4), UNSIGNED(4), or REAL(4) vectors contain 4 elements.

For INTEGER, UNSIGNED and REAL vectors, XL Fortran sets the default kind type parameter of the elements to 4.

Vectors must be aligned on a 16-byte boundary. XL Fortran automatically aligns vectors to 16 bytes, except in the following cases, where:

- The vector is a component of a sequence type or a record structure.
- The vector is a component of a derived type that has the BIND attribute and you compile with the -qalign=bindc=packed or -qalign=bindc=bit_packed options. This aligns the vector to a one-byte boundary.
- The vector is a member of a common block.
- The vector is storage associated with a member of a common block that does not have a 16-byte boundary alignment.

Use the Vector Interlanguage Interoperability table to determine the corresponding XL C/C++ vector type when passing vectors between XL C/C++ and XL Fortran.

<table>
<thead>
<tr>
<th>XL Fortran vector type</th>
<th>XL C/C++ vector type</th>
</tr>
</thead>
<tbody>
<tr>
<td>VECTOR(INTEGER(1))</td>
<td>vector signed char</td>
</tr>
<tr>
<td>VECTOR(INTEGER(2))</td>
<td>vector signed short</td>
</tr>
<tr>
<td>VECTOR(INTEGER(4))</td>
<td>vector signed int, vector signed long</td>
</tr>
<tr>
<td>VECTOR(PIXEL)</td>
<td>vector pixel</td>
</tr>
<tr>
<td>VECTOR-REAL(4)</td>
<td>vector float</td>
</tr>
<tr>
<td>VECTOR-UNSIGNED(1)</td>
<td>vector unsigned char</td>
</tr>
<tr>
<td>VECTOR-UNSIGNED(2)</td>
<td>vector unsigned short</td>
</tr>
<tr>
<td>VECTOR-UNSIGNED(4)</td>
<td>vector unsigned int, vector unsigned long</td>
</tr>
</tbody>
</table>

End of IBM Extension

### Pixel

The PIXEL keyword specifies the pixel type. A pixel is a two-byte entity that the compiler interprets in four parts. The first part consists of one bit. The remaining three parts consist of 5 bits each. You must specify a pixel as part of a vector declaration.

### Unsigned

The UNSIGNED keyword specifies the unsigned integer type. Use the -qintsize compiler option to change the default integer size to 2, 4, or 8 bytes. The default kind type parameter is 4. Unsigned integer literals are not supported. You must specify the unsigned integer type as part of a vector declaration.
Complex

A complex type specifier must include either:
- the COMPLEX keyword, or
- in XL Fortran, the DOUBLE COMPLEX keyword

See “COMPLEX” on page 266 and “DOUBLE COMPLEX” on page 283 for details on declaring entities of type complex.

IBM Extension

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:

<table>
<thead>
<tr>
<th>Kind Type Parameter i</th>
<th>Length Specification j</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPLEX(i)</td>
<td>COMPLEX*j</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>32</td>
</tr>
</tbody>
</table>

End of IBM Extension

The kind of a complex constant is determined by the kind of the constants in the real and imaginary parts in all Fortran compilers.

IBM Extension

In XL Fortran, 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 precision of DOUBLE COMPLEX values is twice that of default complex values.

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

```
(expression, expression)
```

A complex literal constant is a complex constructor where each expression is a pair of initialization expressions. Variables and expressions can be used in each part of the complex constructor as an XL Fortran extension.

End of IBM Extension

Fortran 95

In Fortran 95 you are only allowed to use a single signed integer, or real literal constant 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.

See “COMPLEX” on page 266 and “DOUBLE COMPLEX” on page 283 for details on declaring entities of type complex.

---

**IBM Extension**

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...
- seeeeeeeefffffffffffffffffffffffffffffffffffffffffffffffffffffffff

**COMPLEX(8)** (equivalent to COMPLEX*16)

- Bit no. 0....|....1....|....2....|....3....|....4....|....5....|....6...
- seeeeeeeefffffffffffffffffffffffffffffffffffffffffffffffffffffffff

**COMPLEX(16)** (equivalent to COMPLEX*32)

- Bit no. 0....|....1....|....2....|....3....|....4....|....5....|....6...
- seeeeeeeefffffffffffffffffffffffffffffffffffffffffffffffffffffffff

---

**Examples of complex constants**

**Example 1:**

\[(3_2,-1.86)\]  ! Integer constant 3 is converted to default real

! for constant 3.0.

---

**IBM Extension**

**Example 2:**

\[(4506,6045)\]  ! The imaginary part is converted to extended

! precision 6.4Q5.

---

**Example 3:**

\[(1+1,2+2)\]  ! Use of constant expressions. Both parts are

! converted to default real.

---

End of IBM Extension

---
Logical

IBM Extension

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

<table>
<thead>
<tr>
<th>Kind parameter</th>
<th>Values</th>
<th>Internal (hex) Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.TRUE.</td>
<td>01</td>
</tr>
<tr>
<td></td>
<td>.FALSE.</td>
<td>00</td>
</tr>
<tr>
<td>2</td>
<td>.TRUE.</td>
<td>0001</td>
</tr>
<tr>
<td></td>
<td>.FALSE.</td>
<td>0000</td>
</tr>
<tr>
<td>4</td>
<td>.TRUE.</td>
<td>00000001</td>
</tr>
<tr>
<td></td>
<td>.FALSE.</td>
<td>00000000</td>
</tr>
<tr>
<td>8</td>
<td>.TRUE.</td>
<td>00000000000001</td>
</tr>
<tr>
<td></td>
<td>.FALSE.</td>
<td>0000000000000000</td>
</tr>
</tbody>
</table>

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. Use -qintlog to mix integer and logical data entities in expressions and statements.

The -qport=clogicals option allows you to instruct the compiler to treat all non-zero integers used in logical expressions as TRUE. You must specify both the -qport=clogicals and -qintlog compiler options.

End of IBM Extension

The logical type specifier must include the LOGICAL keyword. See “LOGICAL” on page 346 for details on declaring entities of type logical.

The form of a logical literal constant is:

```
-kind_param
```

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.

IBM Extension

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.

End of IBM Extension

Examples of logical constants

\(.FALSE._4\)
\(.TRUE._4\)

Character

The character type specifier must include the CHARACTER keyword. See “CHARACTER” on page 256 for details on declaring entities of type character.

The form of a character literal constant is:

```
  \[\text{kind}_\text{param}\] \text{character_string}\]
```

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

IBM Extension

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

End of IBM Extension

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, 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.
A zero-length character object uses no storage.

---

**IBM Extension**

In XL Fortran each character object requires 1 byte of storage.

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

<table>
<thead>
<tr>
<th>Escape</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\b</td>
<td>Backspace</td>
</tr>
<tr>
<td>\f</td>
<td>Form feed</td>
</tr>
<tr>
<td>\n</td>
<td>New-line</td>
</tr>
<tr>
<td>\r</td>
<td>New-line</td>
</tr>
<tr>
<td>\t</td>
<td>Tab</td>
</tr>
<tr>
<td>\0</td>
<td>Null</td>
</tr>
<tr>
<td>'</td>
<td>Apostrophe</td>
</tr>
<tr>
<td>&quot;</td>
<td>Double quotation mark</td>
</tr>
<tr>
<td>\</td>
<td>Backslash</td>
</tr>
<tr>
<td>\x</td>
<td>x, where x is any other character</td>
</tr>
</tbody>
</table>

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 XL Fortran Compiler Reference for details and exceptions).

All escape sequences represent a single character.

---

**End of IBM Extension**

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 Compiler Reference) 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.

---

**IBM Extension**

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 XL Fortran Compiler Reference.

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 XL Fortran Compiler Reference for more information.)

**Examples of character constants**

Example 1:
```
'      ! Zero-length character constant.
```

Example 2:
```
1_"ABCDEFGHIJKLMNOPQRSTUVWXYZ"  ! Character constant of length 10, with kind 1.
```

Example 3:
```
'"2"\"A56\"  ! Character constant of length 10 "2'A56\".
```

**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)(scalar_constant, scalar_struct_comp)  (int_expr1 : int_expr2)
```

`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 \( \text{MAX}(\text{int_expr2} - \text{int_expr1} + 1,0) \).

If `int_expr1` is less than or equal to `int_expr2`, their values must be such that:

- \( 1 \leq \text{int_expr1} \leq \text{int_expr2} \leq \text{length} \)

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`.

**IBM Extension**

FORTRAN 77 does not allow character substrings of length 0. Fortran 90 and up does allow these substrings. To perform compile-time checking on substring bounds in accordance with FORTRAN 77 rules, use the `-qzerosize` compiler option. For Fortran 90 compliance, use `-aqzerosize`. To perform run-time checking on substring bounds, use both the `-qcheck` option and the `-qzerosize` (or
-qnozerosize option. (See the XL Fortran Compiler Reference for more information.)

A substring of an array section is treated differently. See “Array sections and substring ranges” on page 77.

Examples of character substrings:

```fortran
CHARACTER(8) ABC, X, Y, Z
ABC = 'ABCD{EFGHIJ}KL'(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 in XL Fortran. See “**BYTE**” on page 249 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. A scalar entity of derived type is called a *structure*. A scalar entity of sequence derived type is called a *sequence structure*. 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.

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

Direct components of a derived type are:

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

Each derived type is resolved into ultimate components of intrinsic data type, allocatable, or pointer.

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 the same as 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, nonallocatable array components must be declared with constant dimension declarators. Pointer and allocatable array components must be declared with a deferred_shape_spec_list.

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 or allocatable.

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.

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

---

IBM Extension

A record structure is a popular extension for manipulating aggregate non-array data. The record structure predates the introduction of derived types in Fortran 90.

The syntax used for record structures parallels that used for Fortran derived types in most cases. Also, in most cases, the semantics of the two features are parallel. For these reasons, record structures are supported in XL Fortran in a way that makes the two features almost completely interchangeable. Hence,

- An entity of a derived type declared using either syntax can be declared using either a TYPE statement or a RECORD statement.
- A component of an object of derived type can be selected using either the percent sign or period.
- A derived type declared using the record structure declaration has a structure constructor.
- A component of any derived type can be initialized using either the standard "equals" form of initialization or the extended "double slashes" form of initialization.

There are differences, however, as outlined here:

- A standard derived type declaration cannot have a %FILL component.
- A record structure declaration must not have a SEQUENCE or PRIVATE statement.
- The \texttt{align=struct} option applies only to derived types declared using a record structure declaration.
- A derived type declared using a record structure declaration may have the same name as an intrinsic type.
- There are differences in the rules for determination of derived types declared using a record structure declaration and those declared using a standard derived type declaration.
- A component of a record structure cannot have the PUBLIC or PRIVATE attribute.
- A derived type declared using the record structure declaration cannot have the BIND attribute.\texttt{F2003 \footnote{F2003}}
The size of a sequence derived type declared using a standard derived type declaration is equal to the sum of the number of bytes required to hold all of its components.

The size of a sequence derived type declared using a record structure declaration is equal to the sum of the number of bytes required to hold all of its components and its padding.

Previously, a numeric sequence structure or character sequence structure that appeared in a common block was treated as if its components were enumerated directly in the common block. Now, that only applies to structures of a type declared using a standard derived type declaration.

```
DERIVED_TYPE_statement
    PRIVATE   ] SEQUENCE ]

component_def_stmt_block

END_TYPE_statement
```

**DERIVED_TYPE_statement**

See “TYPE” on page 404 for details on declaring entities of a specified derived type.

**PRIVATE**

Only one PRIVATE statement can be specified. See “PRIVATE” on page 369 for syntax details.

**SEQUENCE**

Only one SEQUENCE statement can be specified. See “SEQUENCE” on page 396 for syntax details.

**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**, **ALLOCATABLE**, **PRIVATE**, **PUBLIC**, and **POINTER** attributes.

**Fortran 95**

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 409 for detailed syntax and information.

**End of Fortran 95**

**END_TYPE_statement**

can optionally contain the same **type_name** as specified on the TYPE statement. See also “END TYPE” on page 297.

**Fortran 2003 Standard**

You can only specify a PRIVATE statement on a derived-type definition if the
definition is within the specification part of a module. If \texttt{PRIVATE} is used, 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

The default accessibility of a component of a derived type is \texttt{PUBLIC}. To declare the component of a derived type to be private, you can do any of the following:

- Include the \texttt{PRIVATE} statement in the component part of the derived-type definition and do not use the \texttt{PUBLIC} attribute.
- Specify the \texttt{PRIVATE} attribute in the component.
- Declare the derived type itself to be private.

Note that you can only specify the \texttt{PRIVATE} or \texttt{PUBLIC} attribute on a component if the type definition is within the specification part of a module. If a component is private, the component name is accessible only with the module containing the derived type definition, even if the derived type itself is public.

The \texttt{PRIVATE} or \texttt{PUBLIC} attribute must not appear in the components of a record structure.

If a derived type has the \texttt{BIND} attribute, then:

- It cannot be a sequence type.
- It cannot be specified in a \texttt{RECORD} statement.
- Each of its components must be a nonpointer, nonallocatable data component with interoperable type and type parameters.

Note the following rules for working with derived types with the \texttt{BIND} attribute:

- A Fortran derived type is interoperable with a C struct type if:
  - The derived-type definition of the Fortran type specifies the \texttt{BIND} attribute.
  - The Fortran derived type and the C struct type have the same number of components.
  - The components of the Fortran derived type have types and type parameters that are interoperable with the types of the corresponding components of the C struct type.
  - The Fortran type and the C struct type appear in compilation units that are compiled using corresponding alignment options. A component of a Fortran derived type and a component of a C struct type correspond if they are declared in the same relative position in their respective type definitions.

- Derived types with the \texttt{BIND} attribute appearing in compilation units that are compiled with different alignment options are not compatible.
- There is no Fortran type that is interoperable with a C struct type that contains a bit field or that contains a flexible array member.
- There is no Fortran type that is interoperable with a C union type.

End of Fortran 2003 Standard

\section*{Default initialization}

Default initialization may be specified using an equal sign followed by an initialization expression, or by using an \texttt{initial_value_list} enclosed in slashes. You can use this form of initialization for components declared using either a \texttt{record}
structure declaration or a standard derived type declaration.

In Fortran 95 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 nonpointer and nonallocatable 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 nonpointer object or subobject has been specified with default initialization in a type definition, it must not be initialized by a DATA statement.

End of Fortran 95

IBM Extension

A data object of derived type with default initialization can be specified in a common block as an IBM extension. In addition, default initialization does not imply the SAVE attribute in XL Fortran unless -qsave=defaultinit has been specified.

End of IBM Extension

Fortran 95

Unlike explicit initialization, it is not necessary for a data object to have the SAVE attribute for component default initialization to have an effect. You can specify 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.

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 components are either initially defined, or become defined by their corresponding default initialization expressions, and their pointer 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 component will cause the component to:

• become defined, if it is a nonpointer component

• become disassociated, if it is a pointer 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.

End of Fortran 95

Input/Output

Any %FILL field in an entity of derived type is treated as padding in an unformatted data transfer statement.

Determining declared 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:

• They were both declared using standard derived type declarations, both have the same name, either both have the SEQUENCE property, or both have the BIND attribute, and both have components that do not have PRIVATE accessibility and agree in order, name and attributes; or

• They were declared using record structure declarations that were not unnamed, the types have the same name, have no %FILL components and have components that agree in order and attributes, and any %FILL components appear in the same positions in both.

A derived-type definition that has the BIND attribute or the SEQUENCE property 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       ! Sequence derived type
CALL MYSUB(PER1)
PES1 = 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

SUBROUTINE MYPRINT(NAMES) ! External subroutine MYPRINT
  TYPE NAME ! Same type as data type in MYPROG
  ...
  END SUBROUTINE MYPRINT
END SUBROUTINE

An example with different component names

MODULE MOD
  STRUCTURE /S/
    INTEGER I
    INTEGER, POINTER :: P
  END STRUCTURE
  RECORD /S/ R
END MODULE
END PROGRAM

USE MOD, ONLY: R
  STRUCTURE /S/
    INTEGER J
    INTEGER, POINTER :: Q
  END STRUCTURE
  RECORD /S/ R2
  R = R2 ! OK - same type name, components have same attributes and
  ! type (but different names)
END PROGRAM

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 73
and is explained in "Array sections and structure components" on page 77. 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:

**scalar Struct Comp:**

```
name (int_expr_list)
```

*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  
*separator* is `%` or `.`  

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 rightmost *comp_name* must be scalar.

In namelist formatting, a separator must be a percent sign.

If an expression has a form that could be interpreted either as a structure component using periods as separators or as a binary operation, and an operator with that name is accessible in the scoping unit, XL Fortran will treat the expression as a binary operation. If that is not the interpretation you intended, you should use the percent sign to dereference the parts, or, in free source form, insert white space between the periods and the *comp_name*.

**Examples of references to structure components**

**Example 1: Ambiguous use of a period as separator**

```fortran
MODULE MOD
  STRUCTURE /S1/
  STRUCTURE /S2/ BLUE
```
INTEGER I
END STRUCTURE
END STRUCTURE
INTERFACE OPERATOR(.BLUE.)
MODULE PROCEDURE BLUE
END INTERFACE
CONTAINS
INTEGER FUNCTION BLUE(R1, I)
RECORD /S1/ R1
  INTENT(IN) :: R1
  INTEGER, INTENT(IN) :: I
  BLUE = R1%BLUE%I + I
END FUNCTION BLUE
END MODULE MOD

PROGRAM P
USE MOD
RECORD /S1/ R1
  R1%BLUE%I = 17
  I = 13
PRINT *, R1.BLUE.I ! Calls BLUE(R1,I) - prints 30
PRINT *, R1%BLUE%I ! Prints 17
END PROGRAM P

Example 2: Mix of separators

STRUCTURE /S1/
  INTEGER I
END STRUCTURE
STRUCTURE /S2/
  RECORD /S1/ C
END STRUCTURE
RECORD /S2/ R
R.C%I = 17 ! OK
R%C.I = 3 ! OK
R1%C.I = 13 ! OK
R.C.I = 19 ! OK
END

Example 3: Percent and period work for any derived types

STRUCTURE /S/
  INTEGER I, J
END STRUCTURE
TYPE DT
  INTEGER I, J
END TYPE DT
RECORD /S/ R1
TYPE(DT) :: R2
R1.I = 17; R1%J = 13
R2.I = 19; R2%J = 11
END

Allocatable components

 Allocatable components are defined as ultimate components just as pointer components are. This is because the value (if any) is stored separately from the rest of the structure, and this storage does not exist (because the object is unallocated) when the structure is created. As with ultimate pointer components, variables containing ultimate allocatable components are forbidden from appearing directly in input/output lists.
As with allocatable arrays, allocatable components are forbidden from storage association contexts. So, any variable containing an ultimate, allocatable component cannot appear in COMMON or EQUIVALENCE. However, allocatable components are permitted in SEQUENCE types, which allows the same type to be defined separately in more than one scoping unit.

Deallocation of a variable containing an ultimate allocatable component automatically deallocates all such components of the variable that are currently allocated.

In a structure constructor for a derived type containing an allocatable component, the expression corresponding to the allocatable component must be one of the following:

- A reference to the intrinsic function NULL with no argument. The allocatable component receives the allocation status of not currently allocated
- A variable that is itself allocatable. The allocatable component receives the allocation status of the variable and, if it is allocated, the value of the variable. If the variable is an array that is allocated, the allocatable component also has the bounds of the variable
- Any other expression. The allocatable component receives the allocation status of currently allocated with the same value as the expression. If the expression is an array, the allocatable component will have the same bounds.

For intrinsic assignment of those objects of a derived type containing an allocatable component, the allocatable component of the variable on the left-hand-side receives the allocation status and, if allocated, the bounds and value of the corresponding component of the expression. This occurs as if the following sequence of steps is carried out:

1. If the component of the variable is currently allocated, it is deallocated.
2. If the corresponding component of the expression is currently allocated, the component of the variable is allocated with the same bounds. The value of the component of the expression is then assigned to the corresponding component of the variable using intrinsic assignment.

An allocated ultimate allocatable component of an actual argument that is associated with an INTENT(OUT) dummy argument is deallocated on procedure entry so that the corresponding component of the dummy argument has an allocation status of not currently allocated.

This ensures that any pointers that point to the previous contents of the allocatable component of the variable become undefined.

Example:

```fortran
MODULE REAL_POLYNOMIAL_MODULE
  TYPE REAL_POLYNOMIAL
    REAL, ALLOCATABLE :: COEFF(:)
  END TYPE
  INTERFACE OPERATOR(*)
    MODULE PROCEDURE RP_ADD_R, RP_ADD_R
  END INTERFACE
  CONTAINS
  FUNCTION RP_ADD_R(P1,R)
    TYPE(REAL_POLYNOMIAL) RP_ADD_R, P1
    REAL R
    INTENT(IN) P1,R
    ALLOCATE(RP_ADD_R%COEFF(SIZE(P1%COEFF)))
    RP_ADD_R%COEFF = P1%COEFF
  END FUNCTION
END MODULE
```

Chapter 3. Data types and data objects 37
RP_ADD_R%COEFF(1) = P1%COEFF(1) + R
END FUNCTION
FUNCTION RP_ADD_RP(P1,P2)
  TYPE(REAL_POLYNOMIAL) RP_ADD_RP, P1, P2
  INTEGER P1, P2
  ALLOCATE(RP_ADD_RP%COEFF(MAX(SIZE(P1%COEFF), SIZE(P2%COEFF))))
  M = MIN(SIZE(P1%COEFF), SIZE(P2%COEFF))
  IF(M< SIZE(P1%COEFF)>M) THEN
    RP_ADD_RP%COEFF(M+1:) = P1%COEFF(M+1:)
  ELSE IF (SIZE(P2%COEFF)>M) THEN
    RP_ADD_RP%COEFF(M+1:) = P2%COEFF(M+1:)
  END IF
END FUNCTION
END MODULE

PROGRAM EXAMPLE
  USE REAL_POLYNOMIAL_MODULE
  TYPE(REAL_POLYNOMIAL) P, Q, R
  P = REAL_POLYNOMIAL((/4,2,1/)) ! Set P to (X**2+2X+4)
  Q = REAL_POLYNOMIAL((/1,1/)) ! Set Q to (X+1)
  R = P + Q ! Polynomial addition
  PRINT *, 'Coefficients are: ', R%COEFF
END

Structure constructor

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

```fortran
>>> type_name((--))
```

- `type_name` is the name of the derived type.

IBM Extension

If a derived type is declared using the `record structure` declaration and has any `%FILL` component, the structure constructor for that type cannot be used.

If a derived type is accessible in a scoping unit and there is a local entity of class 1 that is not a derived type with the same name accessible in the scoping unit, the structure constructor for that type cannot be used in that scope.

Examples of Structure Constructors

Example 1:

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

---

End of Fortran 2003 Standard

End of IBM Extension

---

Structure constructor

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

```fortran
>>> type_name((--))
```

- `type_name` is the name of the derived type.

IBM Extension

If a derived type is declared using the `record structure` declaration and has any `%FILL` component, the structure constructor for that type cannot be used.

If a derived type is accessible in a scoping unit and there is a local entity of class 1 that is not a derived type with the same name accessible in the scoping unit, the structure constructor for that type cannot be used in that scope.

Examples of Structure Constructors

Example 1:

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

SUBROUTINE TEST2
  TYPE T
    INTEGER EMP_NO
    CHARACTER, POINTER :: EMP_NAME(:)
  END TYPE T
  TYPE (T) EMP_REC
    CHARACTER, TARGET :: NAME(10)
  EMP_REC = T(24744,NAME)
  ! Pointer assignment occurs
END SUBROUTINE
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

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

End of Fortran 95
Record structures

**Declaring record structures**

Declaring a record structure declares a user-defined type in the same way that a standard Fortran derived type definition declares a user-defined type. A type declared using a record structure declaration is a derived type. For the most part, rules that apply to derived types declared using the standard Fortran syntax apply to derived types declared using the record structure syntax. In those cases where there is a difference, the difference will be called out by referring to the two as derived types declared using a record structure declaration and derived types declared using a standard derived type declaration.

**Record structure** declarations follow this syntax:

\[
\text{record\_structure\_dcl}:
\]

\[
\text{structure\_stmt} \\
\quad \text{struct\_comp\_dcl\_item} \\
\quad \text{end\_structure\_stmt}
\]

**struct\_comp\_dcl\_item:**

\[
\text{component\_def\_stmt} \\
\quad \text{record\_structure\_dcl} \\
\quad \text{parameter\_stmt}
\]

where **component\_def\_stmt** is a type declaration statement used to define the components of the derived type.

**structure\_stmt:**

\[
\text{STRUCTURE} \\
\quad /\text{structure\_name}/ \quad /\text{component\_dcl\_list}/
\]

**component\_dcl:**

\[
\quad a \text{-array\_spec-}
\]

where \(a\) is an object name.

A structure statement declares the **structure\_name** to be a derived type in the scoping unit of the nearest enclosing program unit, interface body or subprogram. The derived type is a local entity of class 1 in that scoping unit.
A structure statement may not specify a component_dcl_list unless it is nested in another record structure declaration. Likewise, the structure_name of a structure statement cannot be omitted unless it is part of a record_structure_dcl that is nested in another record structure declaration. A record_structure_dcl must have at least one component.

A derived type declared using a record structure declaration is a sequence derived type, and is subject to all rules that apply to sequence derived types. A component of a type declared using a record structure declaration cannot be of a nonsequence derived type, as is true of sequence derived types declared using standard derived type declarations. A record structure declaration cannot contain a PRIVATE or SEQUENCE statement.

A record structure declaration defines a scoping unit. All statements in the record_structure_dcl are part of the scoping unit of the record structure declaration, with the exception of any other record_structure_dcl contained in the record_structure_dcl. These rules are also true of standard derived type declarations, repeated here for clarity.

A parameter_stmt in a record_structure_dcl declares named constants in the scoping unit of the nearest enclosing program unit, interface body or subprogram. A named constant declared in such a parameter_stmt may have the same name as a component declared in the record_structure_dcl in which it is contained.

Any components declared on a structure_stmt are components of the enclosing derived type, and are local entities of the enclosing structure’s scoping unit. The type of such a component is the derived type on whose structure_stmt it is declared.

Unlike derived types declared using a standard derived type declaration, a derived type name declared using a record structure declaration may be the same as the name of an intrinsic type.

In place of the name of a component, %FILL can be used in a component_def_stmt in a record structure declaration. A %FILL component is used as a place-holder to achieve desired alignment of data in a record structure declaration. Initialization cannot be specified for a %FILL component. Each instance of %FILL in a record structure declaration is treated as a unique component name, different from the names of all other components you specified for the type, and different from all other %FILL components. %FILL is a keyword and is not affected by the -qmixed compiler option.

Each instance of a nested structure that has no name is treated as if it had a unique name, different from the names of all other accessible entities.

As an extension to the rules described on derived types thus far, the direct components of a derived type declared using a record structure declaration are:
• the components of that type that are not %FILL components; and
• the direct components of a derived type component that does not have the POINTER attribute and is not a %FILL component.

The non-filler ultimate components of a derived type are the ultimate components of the derived type that are also direct components.
An object of a derived type with default initialization can be a member of a common block. You must ensure that a common block is not initialized in more than one scoping unit.

Examples of declaring record structures:

Example 1: Nested record structure declarations - named and unnamed

```fortran
STRUCTURE /S1/  
    STRUCTURE /S2/ A ! A is a component of S1 of type S2  
        INTEGER I  
    END STRUCTURE  
    STRUCTURE B ! B is a component of S1 of unnamed type  
        INTEGER J  
    END STRUCTURE  
END STRUCTURE
```

```fortran
RECORD /S1/ R1  
RECORD /S2/ R2 ! Type S2 is accessible here.  
R2.I = 17  
R1.A = R2  
R1.B.J = 13  
END
```

Example 2: Parameter statement nested in a structure declaration

```fortran
INTEGER I  
STRUCTURE /S/  
    INTEGER J  
    PARAMETER(I=17, J=13) ! Declares I and J in scope of program unit to be named constants  
END STRUCTURE  
INTEGER J ! Confirms implicit typing of named constant J  
RECORD /S/ R  
R.J = I + J  
PRINT *, R.J ! Prints 30  
END
```

Example 3: %FILL fields

```fortran
STRUCTURE /S/  
    INTEGER I, %FILL, %FILL(2), J  
    STRUCTURE /S2/ R1, %FILL, R2  
    INTEGER I  
END STRUCTURE  
END STRUCTURE  
RECORD /S/ R  
PRINT *, LOC(R%J)-LOC(R%I) ! Prints 24 with -qintsize=4  
PRINT *, LOC(R%R2)-LOC(R%R1) ! Prints 8 with -qintsize=4  
END
```

Storage mapping

A derived type declared using a `record structure` declaration is a sequence derived type. In memory, objects of such a type will have the components stored in the order specified. The same is true of objects of a sequence derived type declared using a standard derived type declaration.

The `-qalign` option specifies the alignment of data objects in storage, which avoids performance problems with misaligned data. Both the `[no]4k` and `struct` suboptions can be specified and are not mutually exclusive. The default setting is `-qalign=no4k:struct=natural`. `[no]4K` is useful primarily in combination with logical volume I/O and disk striping.

End of IBM Extension
Union and map

A union declares a group of fields in the enclosing record structure that can share the data area in a program.

Unions and maps follow this syntax:

union_dcl:

```
UNION union_dcl_item
END UNION
```

union_dcl_item:

```
map_dcl:
```

map_dcl:

```
MAP map_dcl_item
END MAP
```

map_dcl_item:

```
struct_comp_dcl_item:
```

struct_comp_dcl_item:

```
component_def_stmt
record_structure_dcl
parameter_stmt
union_dcl
```

A union declaration must be defined in a record structure, may be in a map declaration, and a map declaration must be in a union declaration. All declarations in a map_dcl_item within a union declaration must be of the same nesting level,
regardless of which map_dcl they reside in. Therefore, no component name inside a map_dcl may appear in any other map_dcl on the same level.

A component declared within a map declaration must not have a POINTER, PRIVATE, PUBLIC, or ALLOCATABLE attribute.

A record structure with union map must not appear in I/O statements.

The components declared in a map declaration share the same storage as the components declared in the other map declarations within a union construct. When you assign a value to one component in one map declaration, the components in other map declarations that share storage with this component may be affected.

The size of a map is the sum of the sizes of the components declared within it.

The size of the data area established for a union declaration is the size of the largest map defined for that union.

A parameter_stmt in a map declaration or union construct declares entities in the scoping unit of the nearest enclosing program unit, interface body, or subprogram.

A %FILL field in a map declaration is used as a place-holder to achieve desired alignment of data in a record structure. Other non-filler components or part of the components in other map declarations that share the data area with a %FILL field are undefined.

If default initialization is specified in component_def_stmts in at least one map declaration in a union declaration, the last occurrence of the initialization becomes the final initialization of the components.

If default initialization is specified in one of the union map declarations in a record structure, a variable of that type that will have its storage class assigned by default will be given
  • the static storage class if either the -qsave=defaultinit or -qsave=all option is specified; or
  • the automatic storage class, if the -qnosave option is specified.

At any time, only one map is associated with the shared storage. If a component from another map is referenced, the associated map becomes unassociated and its components become undefined. The map referenced will then be associated with the storage.

If a component of map_dcl is entirely or partially mapped with the %FILL component of the other map_dcl in a union, the value of the overlap portion is undefined unless that component is initialized by default initialization or an assignment statement.

Examples of union and map
Example 1: The size of the union is equal to the size of the largest map in that union

structure /S/
union
map
  integer*4 i, j, k
  real*8 r, s, t
Example 2: The results of union map are different with different -qsave option and suboptions

PROGRAM P
CALL SUB
CALL SUB
END PROGRAM P

SUBROUTINE SUB
  LOGICAL, SAVE :: FIRST_TIME = .TRUE.
  STRUCTURE /S/
    UNION
      MAP
        INTEGER I/17/
      END MAP
      MAP
        INTEGER J
      END MAP
    END UNION
  END STRUCTURE
  RECORD /S/ LOCAL_STRUCT
  INTEGER LOCAL_VAR

  IF (FIRST_TIME) THEN
    LOCAL_STRUCT.J = 13
    LOCAL_VAR = 19
    FIRST_TIME = .FALSE.
  ELSE
    ! Prints "13" if compiled with -qsave or -qsave=all
    ! Prints "13" if compiled with -qsave=defaultinit
    ! Prints "17" if compiled with -qnosave
    PRINT *, LOCAL_STRUCT%j
    ! Prints "19" if compiled with -qsave or -qsave=all
    ! Value of LOCAL_VAR is undefined otherwise
    PRINT *, LOCAL_VAR
  END IF
END SUBROUTINE SUB

Example 3: The last occurrence of default initialization in a map declaration within a union structure becomes the final initialization of the component

structure /st/
  union
    map
      integer i /3/, j /4/
    union
      map
        integer k /8/, l /9/
      END MAP
    END UNION
    map
      integer a, b
    union
      map
        integer c /21/
    END MAP
  END UNION
Example 4: The following program is compiled with -qintsize=4 and -qalign=struct=packed, the components in the union MAP are aligned and packed

```fortran
structure /s/
  union /s/ 
    integer*2  i /z'1a1a'/, %FILL, j /z'2b2b'/
  end map 
  integer  m, n 
  end map 
end union
end structure
record /s/ r

print '(2z6.4)', r.i, r.j ! Prints "1A1A 2B2B"
print '(2z10.8)', r.m, r.n ! Prints "1A1A0000 2B2B0000" however ! the two bytes in the lower order are ! not guaranteed.
  r.m = z'abc00cba' ! Components are initialized by ! assignment statements.
  r.n = z'02344320'
print '(2z10.8)', r.m, r.n ! Prints "ABC00CBA 02344320"
p *, R.i, R.j, R.k, R.l ! Prints "3 4 21 9"
p *, R.a, R.b, R.c ! Prints "3 4 21"
end
```

---

**Typeless literal constants**

A typeless constant does not have an intrinsic type in XL Fortran. 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:

```
X'hexadecimal_number'
Z"hexadecimal_number"
```

---

**End of IBM Extension**
hexadecimal_number

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

The Znn...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 \texttt{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:

\[
\begin{align*}
0 & \text{'--octal\_number--'} \\
'--octal\_number--' & 0 \\
'--octal\_number--' & '--octal\_number--'
\end{align*}
\]

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 \texttt{INTEGER(2)} data object can be represented by a 6-digit octal constant if the leftmost digit is 0 or 1; an \texttt{INTEGER(4)} data object can be represented by an 11-digit constant if the leftmost digit is 0, 1, 2, or 3; an \texttt{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

0'01234567'  
"01234567"0

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:

```
$nH$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” for information on how the constant is interpreted by XL Fortran.

**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 the data type and length until you use them and context is understood.

- 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), the length is determined by the number of bytes represented by the typeless constant.
- 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. However, if you specify the compiler option `-qport=typlessarg`, the actual argument is converted to default integer. 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:

```fortran
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. However, using the compiler option `-qctyplessarg` will cause a Hollerith constant to be passed as if it were an integer actual argument. See §6.2.1.4 Resolution of procedure references on page 169 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 169 for more information about establishing a procedure name to be generic or specific.

• When you use a typeless constant as a selector in the ASSOCIATE statement, the constant assumes default integer type.
• 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.
• A kind type parameter must not be replaced with a logical constant even if -qintlog is on, nor by a character constant even if -qctypless is on, nor can it be a typeless constant.

Examples of typeless constants in expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT=B'1'</td>
<td>Binary constant is default integer</td>
</tr>
<tr>
<td>RL4=X'1'</td>
<td>Hexadecimal constant is default real</td>
</tr>
<tr>
<td>INT=INT + O'1'</td>
<td>Octal constant is default integer</td>
</tr>
<tr>
<td>RL4=INT + B'1'</td>
<td>Binary constant is default integer</td>
</tr>
<tr>
<td>INT=RL4 + Z'1'</td>
<td>Hexadecimal constant is default real</td>
</tr>
<tr>
<td>ARRAY(0'1')=1.0</td>
<td>Octal constant is default integer</td>
</tr>
<tr>
<td>LOGICAL(8) LOG8</td>
<td>Logical constant is LOGICAL(8), LOG8 is .TRUE.</td>
</tr>
</tbody>
</table>

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 409 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 326 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 either 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, complex or derived-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.
- A derived type scalar object is defined if and only if all of its nonpointer components are defined.
- 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. Variables with the BIND attribute that are initialized by means other than Fortran are also initially defined. In addition, default initialization can 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 statement 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.

---

**Fortran 95**

5. Default initialization may cause a variable to be initially defined.

---

**End of Fortran 95**

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 dummy argument does not have **INTENT(OUT)**, and 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 that does not have **INTENT(OUT)** 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.

---

**Fortran 2003 Standard**

10. Execution of an input/output statement containing an **IOMSG**= specifier causes the specified character variable to become defined when an error, end-of-file or end-of-record occurs.

---

**End of Fortran 2003 Standard**

11. Execution of a **READ** statement containing a **SIZE**= specifier causes the specified integer variable to become defined.

12. Execution of a **READ** or **WRITE** statement in XL Fortran containing an **ID**= specifier causes the specified integer variable to become defined.

13. Execution of a **WAIT** statement in XL Fortran containing a **DONE**= specifier causes the specified logical variable to become defined.

14. Execution of a synchronous **READ** or **WRITE** statement in XL Fortran 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.

15. 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.

16. 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.

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

18. 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.

19. 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.

20. Execution of an ALLOCATE or DEALLOCATE statement with a STAT= specifier causes the variable specified by the STAT= specifier to become defined.

21. Allocation of a zero-sized array causes the array to become defined.

22. Invocation of a procedure causes any automatic object of zero size in that procedure to become defined.

23. Execution of a pointer assignment statement that associates a pointer with a target that is defined causes the pointer to become defined.

24. Invocation of a procedure that contains a nonpointer, nonallocatable, automatic object, causes all nonpointer default-initialized subcomponents of the object to become defined.

25. Invocation of a procedure that contains a nonpointer nonallocatable INTENT(OUT) dummy argument causes all nonpointer default-initialized subcomponents of the object to become defined.

26. Allocation of an object of a derived type where a nonpointer component is initialized by default initialization, causes the component and its subobjects to become defined.

---

**Fortran 95**

27. In a FORALL statement or construct used in Fortran 95, the index-name becomes defined when the index-name value set is evaluated.

---

**IBM Extension**

28. If a THREADPRIVATE nonpointer nonallocatable variable that does not appear in a COPYIN clause is defined on entry into the first parallel region, each new thread’s copy of the variable is defined.

29. If a THREADPRIVATE common block that does not appear in a COPYIN clause is defined on entry into the first parallel region, each new thread’s copy of the variable is defined.

30. For THREADPRIVATE variables that are specified in a COPYIN clause, each new thread duplicates the master thread’s definition, allocation and association status of these variables. 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.
31. For THREADPRIVATE common blocks that are in a COPYIN clause, each new thread duplicates the master thread’s definition, allocation and association status of the variables in these common blocks. Therefore, if the master thread’s copy of a common block variable is defined on entry to a parallel region, each new thread’s copy of the common block variable will also be defined.

32. When a variable is specified in a FIRSTPRIVATE clause of a PARALLEL, PARALLEL DO, DO, PARALLEL SECTIONS, PARALLEL WORKSHARE, 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.

33. For each variable, or variable inside a common block, specified in a COPYPRIVATE clause, then after the execution of the code enclosed in the SINGLE construct and before any threads in the team have left the construct, all copies of the variable become defined as follows:

- If the variable has the POINTER attribute, then copies of the variable in other threads in the team have the same pointer association status as the copy of the variable belonging to the thread that executed the code enclosed in the SINGLE construct.
- If the variable does not have the POINTER attribute, then copies of the variable in other threads in the team have the same definition as the copy of the variable belonging to the thread that executed the code enclosed in the SINGLE construct.

Events causing undefined

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=, NEXTREC=, and POS= 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 undefined 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 entity is deallocated, it becomes undefined. Successful execution of an ALLOCATE statement for a nonzero-sized object for which default initialization has not been specified causes the object 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= or IOMSG= 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 nonpointer dummy argument with \texttt{INTENT(OUT)} and its associated actual argument are undefined, except for nonpointer direct components that have default initialization.

c. A pointer dummy argument with \texttt{INTENT(OUT)} and its associated actual argument have an association status of 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 \texttt{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.

\begin{center}
\textbf{Fortran 95}
\end{center}

16. When the execution of a \texttt{FORALL} statement or construct in Fortran 95 has completed, the \textit{index-name} becomes undefined.

\begin{center}
\textbf{End of Fortran 95}
\end{center}

\begin{center}
\textbf{Fortran 2003 Standard}
\end{center}

17. When execution of a \texttt{RETURN} or \texttt{END} statement causes a variable to become undefined, any variable of type \texttt{C_PTR} becomes undefined if its value is the C address of any part of the variable that becomes undefined.

18. When a variable with the \texttt{TARGET} attribute is deallocated, any variable of type \texttt{C_PTR} becomes undefined if its value is the C address of any part of the variable that is deallocated.

\begin{center}
\textbf{End of Fortran 2003 Standard}
\end{center}

\begin{center}
\textbf{IBM Extension}
\end{center}

19. When a variable is specified in either the \texttt{PRIVATE} or \texttt{LASTPRIVATE} clause of a \texttt{PARALLEL}, \texttt{PARALLEL DO}, \texttt{DO}, \texttt{PARALLEL SECTIONS}, \texttt{PARALLEL WORKSHARE}, \texttt{SECTIONS} or \texttt{SINGLE} directive, each new thread’s copy of the variable is undefined when the thread is first created.

20. When a variable is specified in a \texttt{FIRSTPRIVATE} clause of a \texttt{PARALLEL}, \texttt{PARALLEL DO}, \texttt{DO}, \texttt{PARALLEL SECTIONS}, \texttt{PARALLEL WORKSHARE}, \texttt{SECTIONS} or \texttt{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.

21. When a variable is specified in the \texttt{NEW} clause of an \texttt{INDEPENDENT} directive, the variable is undefined at the beginning of every iteration of the following \texttt{DO} loop.

22. When a variable appears in asynchronous input, that variable becomes undefined, and remains undefined, until the matching \texttt{WAIT} statement is reached.

23. If a \texttt{THREADPRIVATE} common block or a \texttt{THREADPRIVATE} variable is specified in a \texttt{COPYIN} clause, each new thread duplicates the master thread’s definition, allocation and association status of the variables. 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.
24. If a THREADPRIVATE common block variable or a THREADPRIVATE variable has the ALLOCATABLE attribute, the allocation status of each copy created will be not currently allocated.

25. If a THREADPRIVATE common block variable or a THREADPRIVATE variable has the POINTER attribute with an initial association status of disassociated through either default or explicit initialization, each copy will have an association status of disassociated. Otherwise the association status of each copy is undefined.

26. If a THREADPRIVATE common block variable or a THREADPRIVATE variable has neither the ALLOCATABLE nor the POINTER attribute and is initially defined through default or explicit initialization, each copy has the same definition. Otherwise, each copy is undefined.

### Allocation status

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

- Not currently allocated, which means that the object has never been allocated or that the last operation on it was a deallocation.
- Currently allocated, which means that the object has been allocated by an ALLOCATE statement and has not been subsequently deallocated.
- Undefined, which means that the object 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.

---

**IBM Extension**

In XL Fortran, this status is only available when you are using the `-qxlf90=noautodealloc` option. (For example, you are using the `xlif90` compilation command.)

---

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

When the allocation status of an allocatable object changes, the allocation status of any associated allocatable object changes accordingly.

---

**Fortran 95**

In Fortran 95, the allocation status of an allocatable object 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.

---

End of Fortran 95
In XL Fortran, the allocation status of such an object remains currently allocated.

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. 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 objects. 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 161.

Storage class assignment

Variable names are assigned storage classes in one of the following ways:

1. Explicitly:
   - Dummy arguments have an explicit storage class of reference parameter or value parameter. See “%VAL and %REF” on page 161 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 objects 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 or,
• The variable is declared in the specification part of a main program.

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=all** compiler option.
   • Static, for variables of derived type that have default initialization specified, and automatic otherwise if you specify the **-qsave=defaultinit** compiler option.
   • Automatic, if you specified the **-qnosave** compiler option. This is the default setting.
See `-qsave option` in the `XL Fortran Compiler Reference` for details on the default settings with regard to the invocation commands.

End of IBM Extension

---
Chapter 4. Array concepts

XL Fortran provides a set of features, commonly referred to as array language, that allow you to manipulate arrays. This section 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 67
- “Deferred-shape arrays” on page 68
- “Assumed-size arrays” on page 70
- “Array elements” on page 72
- “Array sections” on page 73
- “Array constructors” on page 79
- “Expressions involving arrays” on page 81

Related information:
- Many statements in Chapter 10, “Statements and attributes,” on page 235, have special features and rules for arrays.
- This section makes frequent use of the DIMENSION attribute. See “DIMENSION” on page 279.
- A number of intrinsic functions are especially for arrays. These functions are mainly those classified as “Transformational intrinsic functions” on page 474.

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.

IBM Extension

In XL Fortran, the bound of a dimension can be positive, negative or zero within the range -(2^31) to 2^31-1 in 32–bit mode. The range for bounds in 64-bit mode is -(2^63) to 2^63-1.

End of IBM Extension
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 return values of the intrinsic inquiry functions **LBOUND** and **UBOUND** for 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 86
- "**LBOUND(ARRAY, DIM)**" on page 535
- "**UBOUND(ARRAY, DIM)**" on page 603

**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.

```fortran
INTEGER, DIMENSION(5) :: X ! 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.

---

**IBM Extension**

The theoretical maximum number of elements in an array is \(2^{31}-1\) elements in 32-bit mode, or \(2^{63}-1\) elements in XL Fortran 64-bit mode. Hardware addressing considerations make it impractical to declare any combination of data objects whose total size (in bytes) exceeds this value.

---

**End of IBM Extension**

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:

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

Standard Fortran allows an array to have up to seven dimensions.

---

**IBM Extension**

An array can have from one to twenty dimensions in XL Fortran.

---

**End of IBM Extension**

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:

\[
\begin{align*}
\text{INTEGER, DIMENSION (10,10) :: A} & \quad \text{! Shape = (/ 10, 10 /)} \\
\text{REAL, DIMENSION (-5:4,1:10,10:19) :: B} & \quad \text{! Shape = (/ 10, 10, 10 /)}
\end{align*}
\]

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

\[
\begin{align*}
\text{INTEGER A(5)} & \quad \text{! Size = 5} \\
\text{REAL B(-1:0,1:3,4)} & \quad \text{! Size = 2 * 3 * 4 = 24}
\end{align*}
\]

**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 585 and SIZE(ARRAY, DIM) on page 589. 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 any of the Compatible Statements and Attributes for Array Declarators table.

**Table 8. Compatible statements and attributes for array declarators**

<table>
<thead>
<tr>
<th>ALLOCATABLE</th>
<th>AUTOMATIC</th>
<th>COMMON</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIMENSION</td>
<td>PARAMETER</td>
<td>POINTER (integer)</td>
</tr>
<tr>
<td>POINTER</td>
<td>PROTECTED</td>
<td>STATIC</td>
</tr>
<tr>
<td>TARGET</td>
<td>Type Declaration</td>
<td>VOLATILE</td>
</tr>
</tbody>
</table>

For example:

\[
\begin{align*}
\text{DIMENSION :: A(1:5)} & \quad \text{! Declarator is "(1:5)"} \\
\text{REAL, DIMENSION(1,1:5) :: B} & \quad \text{! Declarator is "(1,1:5)"} \\
\text{INTEGER C(10)} & \quad \text{! Declarator is "(10)"}
\end{align*}
\]

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_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.

```
Explicit_shape_spec_list
```

- `lower_bound`, `upper_bound`
  - are specification expressions

If any bound is not constant, the array must be declared inside a subprogram. 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.

The size of an explicit-shape dummy argument does not need to be the same as
the size of the actual argument, but the size of the dummy argument cannot be
larger than the size of the actual argument.

### 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(S)
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

END SUBROUTINE

Related information
• For general information about automatic data objects, see “Automatic objects” on page 16 and “Storage classes for variables” on page 59.

Adjustable arrays
An adjustable array is an explicit-shape array dummy argument that has at least one non-constant bound.

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 must be declared in integer POINTER 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 bounds are evaluated on entry to the subprogram, and remain constant during execution of the subprogram.

With the -qddim compiler option, as explained in the XL Fortran Compiler Reference, 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 elements 2 through 6 of ARRAY
PRINT *, PTE
N = 7 ! Increase the size
PRINT *, PTE
END

Related information:
“POINTER (integer)” on page 366

End of IBM Extension

Assumed-shape arrays
Assumed-shape arrays are dummy argument arrays where the extent of each dimension is taken from the associated actual arguments.
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**

```fortran
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 can 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 the ALLOCATABLE or POINTER attribute, while an assumed-shape array must be a dummy argument that does not have the ALLOCATABLE or 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 112
- “Pointer association” on page 136
- “ALLOCATABLE” on page 238
- “ALLOCATED(X)” on page 485
- “ASSOCIATED(POINTER, TARGET)” on page 488

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
```
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.

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

Related information:
• "Pointer association" on page 136

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.

Related information:
• "Allocation status" on page 58
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:

\[
\text{MAX} \left( \text{INT} \left( \frac{T - A + 1}{S} \right), 0 \right)
\]

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 7
CALL SUB1(A(6)(5:10)) ! Size of dummy argument array is 4 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

```fortran
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,:)
CALL SUB1(X)
END
SUBROUTINE SUB1(Y)
INTEGER Y(2,*)
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
  ! dimension, but not the last one.
END SUBROUTINE
SUBROUTINE SUB2(Y)
INTEGER Y(*)
PRINT *, Y(6) ! Y has a different rank than X above.
  ! We have to know (or compute) the position of
  ! the last element. Nothing prevents us from
  ! subscipting 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 `PARAMETER`, `PROTECTED`, `TARGET`, and `VOLATILE` 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(array_struct_comp)subscript_list
```
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 scalar integer expression

--- IBM Extension ---

A subscript can be a real expression in XL Fortran.

--- End of IBM Extension ---

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 34
“Array sections and structure components” on page 77

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:

<table>
<thead>
<tr>
<th>Position of Array Element</th>
<th>Array Element Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(1,1,1)</td>
<td>1</td>
</tr>
<tr>
<td>A(2,1,1)</td>
<td>2</td>
</tr>
<tr>
<td>A(1,2,1)</td>
<td>3</td>
</tr>
<tr>
<td>A(2,2,1)</td>
<td>4</td>
</tr>
<tr>
<td>A(1,3,1)</td>
<td>5</td>
</tr>
<tr>
<td>A(2,3,1)</td>
<td>6</td>
</tr>
<tr>
<td>A(1,1,2)</td>
<td>7</td>
</tr>
<tr>
<td>A(2,1,2)</td>
<td>8</td>
</tr>
<tr>
<td>A(1,2,2)</td>
<td>9</td>
</tr>
<tr>
<td>A(2,2,2)</td>
<td>10</td>
</tr>
<tr>
<td>A(1,3,2)</td>
<td>11</td>
</tr>
<tr>
<td>A(2,3,2)</td>
<td>12</td>
</tr>
</tbody>
</table>

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 77 explains the additional rules for specifying array sections that are also structure components.

```
array_name (section_subscript_list) substring_range
```

**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 expression, explained in “Array elements” on page 72.

```
IBM Extension
```

A subscript can be a real expression in XL Fortran.

```
End of IBM Extension
```

**subscript_triplet, vector subscript**

designate a (possibly empty) sequence of subscripts in a given dimension. For details, see “Subscript triplets” on page 73 and “Vector subscripts” on page 76.

**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 : int_expr2
```

`int_expr1` and `int_expr2` are scalar integer expressions called substring expressions, defined in “Character substrings” on page 27. 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 \( \text{SECTION} = A(1:3, (/ 5,6,5 /), 4) \):
- 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:

\[
\begin{align*}
A(1,5,4) & \quad | \quad \text{SECTION}(1,1) \\
A(2,5,4) & \quad | \quad \text{SECTION}(2,1) \\
A(3,5,4) & \quad | \quad \text{SECTION}(3,1) \\
A(1,6,4) & \quad | \quad \text{SECTION}(1,2) \\
A(2,6,4) & \quad | \quad \text{SECTION}(2,2) \\
A(3,6,4) & \quad | \quad \text{SECTION}(3,2) \\
A(1,5,4) & \quad | \quad \text{SECTION}(1,3) \\
A(2,5,4) & \quad | \quad \text{SECTION}(2,3) \\
A(3,5,4) & \quad | \quad \text{SECTION}(3,3)
\end{align*}
\]

Some examples of array sections include:

```fortran
INTEGER, DIMENSION(20,20) :: A
!
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 34.

## 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.

\[
\text{subscript1, subscript2 : stride}
\]

**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 expression that specifies how many subscript positions to count to reach the next selected element.
A stride can be a real expression in XL Fortran. 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 120.

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(1:9:2) ! 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(10:10:-1) ! both zero-sized.
END
```

**Vector subscripts**

A vector subscript is an integer 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:

```fortran
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.

```fortran
! 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.

```fortran
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

Understanding how array sections and structure components interact requires a familiarity with the syntax for ["Structure components" on page 34.](#)

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:
\textit{object\_name}

is the name of an object of derived type

\textit{section\_subscript\_list, substring\_range}

are the same as defined under \textbf{“Array sections” on page 73}

\textit{comp\_name}

is the name of a derived-type component

\% or . Separator character.

\textbf{Note:} The . (period) separator is an IBM extension.

\textbf{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 \textit{comp\_name} must have a \textit{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 \textbf{ALLOCATABLE} or \textbf{POINTER} attributes.

\begin{verbatim}
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
!

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=1, shape=(/51/).
!
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
\end{verbatim}
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 579 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.

```


```

implodedo_variable

is a named scalar integer or real variable.

In a nonexecutable statement, the type must be integer. You must not
reference the value of an implied_do_variable in the limit expressions expr1
or expr2. Loop processing follows the same rules as for an implied-DO in
"DATA" on page 273, 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 scalar integer or real 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)
```

XL Fortran Language Reference
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
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!

END
```

Related information:

“Elemental intrinsic procedures” on page 473
“Intrinsic assignment” on page 100
“WHERE” on page 423 shows a way to assign values to some elements in an array but not to others
“FORALL construct” on page 109
Chapter 5. Expressions and assignment

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

- “Introduction to expressions and assignment”
- “Constant expressions” on page 84
- “Initialization expressions” on page 85
- “Specification expressions” on page 86
- “Operators and expressions” on page 88
- “Extended intrinsic and defined operations” on page 96
- “How expressions are evaluated” on page 97
- “Intrinsic assignment” on page 100
- “WHERE construct” on page 103
- “FORALL construct” on page 105
- “Pointer assignment” on page 112

Related information

- “Defined operators” on page 146
- “Defined assignment” on page 147

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:

\[ \text{operator operand} \]

A binary operation has the form:

\[ \text{operand}_1 \text{ operator operand}_2 \]

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 operand, and every element of this 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

\[ \frac{I*J}{K} \]
\[ I*(J/K) \]

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.3</td>
<td>Constant</td>
</tr>
<tr>
<td>'ABCD'</td>
<td>Subobject of a constant</td>
</tr>
<tr>
<td>VAR</td>
<td>Variable name</td>
</tr>
<tr>
<td>(/7.0,8.0/)</td>
<td>Array constructor</td>
</tr>
<tr>
<td>EMP(6,'SMITH')</td>
<td>Structure constructor</td>
</tr>
<tr>
<td>SIN(X)</td>
<td>Function reference</td>
</tr>
<tr>
<td>(T-1)</td>
<td>Expression in parentheses</td>
</tr>
</tbody>
</table>

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. A disassociated pointer can also appear as an actual argument to the ASSOCIATED intrinsic inquiry function.

Given the operation \[ \text{op expr1} \text{ op expr2} \], the shape of the operation is the shape of \text{expr2} if \text{op} is unary or if \text{expr1} is a scalar. Otherwise, its shape is that of \text{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 KIND, LEN, or NEW_LINE 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.

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 that is subject to all the same rules. In addition, the following rules apply to items that form primaries for initialization expressions:

• 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.

• You can reference one of the following transformational intrinsic functions, where each argument must be an initialization expression:
  – REPEAT
  – RESHAPE
  – SELECTED_INT_KIND
  – SELECTED_REAL_KIND
  – TRANSFER
  – TRIM
  – IEEE_SELECTED_REAL_KIND

You can also specify the generic intrinsic functions and related specific functions found in the Intrinsic Functions for Initialization Expressions table. Unless otherwise noted, all table entries are provided as IBM extensions, though the intrinsic functions themselves are not extensions.

Table 9. Intrinsic functions for initialization expressions

<table>
<thead>
<tr>
<th><strong>ABS</strong> (ABS, DABS, and QABS specific functions only)</th>
<th><strong>IMAG</strong></th>
<th><strong>NULL</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>AIMAG</td>
<td>INDEX</td>
<td>QCMPLX</td>
</tr>
</tbody>
</table>
Table 9. Intrinsic functions for initialization expressions (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Type 1</th>
<th>Type 2</th>
<th>Type 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPLX</td>
<td>INT</td>
<td>QEXT</td>
<td></td>
</tr>
<tr>
<td>CONJG</td>
<td>MAX</td>
<td>REAL</td>
<td></td>
</tr>
<tr>
<td>DBLE</td>
<td>MIN</td>
<td>SCAN</td>
<td></td>
</tr>
<tr>
<td>DCMPLX</td>
<td>MOD</td>
<td>SIGN</td>
<td></td>
</tr>
<tr>
<td>DIM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QDIM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>DIM, DDIM, and QDIM specific functions only</em></td>
<td>NEW_LINE</td>
<td>VERIFY</td>
<td></td>
</tr>
<tr>
<td>DIM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NINT</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes:
1. Fortran 95
2. Fortran 2003 Standard

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 functions LEN and the kind inquiry function KIND, an IEEE inquiry function, 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.

Fortran 95

• A reference to any remaining intrinsic functions defined in this document where each argument is a restricted expression.

End of Fortran 95

IBM Extension

• A reference to a system inquiry function, where any arguments are restricted expressions.

End of IBM Extension

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

Fortran 95

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.

End of Fortran 95

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

\[
\begin{align*}
\text{LBOUND(C,2)+6} & \quad \text{C is an assumed-shape dummy array} \\
\text{ABS(I)*J} & \quad \text{I and J are scalar integer variables} \\
\text{276/NNS(4)} & \quad \text{NN is accessible through host association}
\end{align*}
\]

Fortran 95

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 contains details on the XL Fortran expressions listed in the *XL Fortran Expressions* table. For information on the order of evaluation precedence see, How expressions are evaluated.

Table 10. XL Fortran expressions

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Logical</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>Primary</td>
</tr>
<tr>
<td>General</td>
<td>Relational</td>
</tr>
</tbody>
</table>

**Arithmetic**

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

```
+   -
```

The form of *arith_term* is:

```
/   *
```

---

End of Fortran 95
The form of \textit{arith\_factor} is:

\begin{verbatim}
&\textit{arith\_primary} \textbf{**} \textit{arith\_factor}
\end{verbatim}

An \textit{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.

<table>
<thead>
<tr>
<th>Arithmetic Operator</th>
<th>Representation</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>Exponentiation</td>
<td>First</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>Second</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>Second</td>
</tr>
<tr>
<td>+</td>
<td>Addition or identity</td>
<td>Third</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction or negation</td>
<td>Third</td>
</tr>
</tbody>
</table>

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 \(-2\).

---

**IBM Extension**

For details of exception conditions that can arise during evaluation of floating-point expressions, see [Detecting and trapping floating-point exceptions](#) in

---

Chapter 5. Expressions and assignment
Examples of arithmetic expressions

<table>
<thead>
<tr>
<th>Arithmetic Expression</th>
<th>Fully Parenthesized Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>-b**2/2.0</td>
<td>-((b**2)/2.0)</td>
</tr>
<tr>
<td>i<strong>j</strong>2</td>
<td>i**(j**2)</td>
</tr>
<tr>
<td>a/b**2 - c</td>
<td>(a/(b**2)) - c</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>first operand</th>
<th>I(1)</th>
<th>I(2)</th>
<th>I(4)</th>
<th>I(8)</th>
<th>R(4)</th>
<th>R(8)</th>
<th>R(16)</th>
<th>X(4)</th>
<th>X(8)</th>
<th>X(16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>I(1)</td>
<td>I(1)</td>
<td>I(2)</td>
<td>I(4)</td>
<td>I(8)</td>
<td>R(4)</td>
<td>R(8)</td>
<td>R(16)</td>
<td>X(4)</td>
<td>X(8)</td>
<td>X(16)</td>
</tr>
<tr>
<td>I(2)</td>
<td>I(2)</td>
<td>I(2)</td>
<td>I(4)</td>
<td>I(8)</td>
<td>R(4)</td>
<td>R(8)</td>
<td>R(16)</td>
<td>X(4)</td>
<td>X(8)</td>
<td>X(16)</td>
</tr>
<tr>
<td>I(4)</td>
<td>I(4)</td>
<td>I(4)</td>
<td>I(4)</td>
<td>I(8)</td>
<td>R(4)</td>
<td>R(8)</td>
<td>R(16)</td>
<td>X(4)</td>
<td>X(8)</td>
<td>X(16)</td>
</tr>
<tr>
<td>I(8)</td>
<td>I(8)</td>
<td>I(8)</td>
<td>I(8)</td>
<td>I(8)</td>
<td>R(4)</td>
<td>R(8)</td>
<td>R(16)</td>
<td>X(4)</td>
<td>X(8)</td>
<td>X(16)</td>
</tr>
<tr>
<td>X(4)</td>
<td>X(4)</td>
<td>X(4)</td>
<td>X(4)</td>
<td>X(4)</td>
<td>X(4)</td>
<td>X(8)</td>
<td>X(16)</td>
<td>X(4)</td>
<td>X(8)</td>
<td>X(16)</td>
</tr>
<tr>
<td>X(8)</td>
<td>X(8)</td>
<td>X(8)</td>
<td>X(8)</td>
<td>X(8)</td>
<td>X(8)</td>
<td>X(8)</td>
<td>X(16)</td>
<td>X(8)</td>
<td>X(8)</td>
<td>X(16)</td>
</tr>
<tr>
<td>X(16)</td>
<td>X(16)</td>
<td>X(16)</td>
<td>X(16)</td>
<td>X(16)</td>
<td>X(16)</td>
<td>X(16)</td>
<td>X(16)</td>
<td>X(16)</td>
<td>X(16)</td>
<td>X(16)</td>
</tr>
</tbody>
</table>

IBM Extension

Notes:

1. XL Fortran implements integer operations using \texttt{INTEGER(4)} arithmetic, or \texttt{INTEGER(8)} arithmetic if data items are 8 bytes in length. If the intermediate result is used in a context requiring \texttt{INTEGER(1)} or \texttt{INTEGER(2)} data type, it is converted as required.

   ```fortran
   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=-2
   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
   ```

End of IBM Extension
Character

A character expression, when evaluated, produces a result of type character. The form of $\text{char}_\text{expr}$ is:

\[
\text{char} \_ \text{primary} \quad \text{char} \_ \text{expr} \quad \text{char} \_ \text{primary}
\]

$\text{char}_\text{primary}$ 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 include concatenation of an operand when you declare the length with an asterisk in parentheses. This indicates inherited length. In this case, the actual length depends on whether you use the inherited length character string 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. The character expression 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
```

General

The general form of an expression ($\text{general}_\text{expr}$) is:

\[
\text{expr} \quad \text{general}_\text{expr} \quad \text{defined}_\text{binary}_\text{op}
\]

$\text{defined}_\text{binary}_\text{op}$ is a defined binary operator. See “Extended intrinsic and defined operations” on page 96.
expr is one of the kinds of expressions defined below.

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

**Logical**

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

\[ \text{logical_expr} \rightarrow \text{logical_secondary} \]

The form of a logical_secondary is:

\[ \text{logical_secondary} \rightarrow \text{logical_disjoint} \]

The form of a logical_disjoint is:

\[ \text{logical_disjoint} \rightarrow \text{logical_term} \]

The form of a logical_term is:

\[ \text{logical_term} \rightarrow \text{logical_factor} \]

The form of a logical_factor is:

\[ \text{logical_factor} \rightarrow \text{logical_primary} \]

logical_primary is a primary of type logical.

rel_expr is a relational expression.

The logical operators are:

<table>
<thead>
<tr>
<th>Logical Operator</th>
<th>Representing</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>.NOT.</td>
<td>Logical negation</td>
<td>First (highest)</td>
</tr>
<tr>
<td>.AND.</td>
<td>Logical conjunction</td>
<td>Second</td>
</tr>
<tr>
<td>.OR.</td>
<td>Logical inclusive disjunction</td>
<td>Third</td>
</tr>
</tbody>
</table>
Logical Operator | Representing | Precedence
---|---|---
.XOR. (See Note *) | Logical exclusive disjunction | Fourth (lowest) (See Note *)
.EQV. | Logical equivalence | Fourth (lowest)
.NEQV. | Logical nonequivalence | Fourth (lowest)

**Note:** * XL Fortran logical operator.

---

**IBM Extension**

The .XOR. operator is treated as an intrinsic operator only when the `-qxlf77=intxor` compiler option is specified. (See the [`-qxlf77` Option](#) in the [XL Fortran Compiler Reference](#) 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.

---

End of IBM Extension

---

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:

<table>
<thead>
<tr>
<th>x1</th>
<th>.NOT. x1</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x1</th>
<th>x2</th>
<th>.AND.</th>
<th>.OR.</th>
<th>.XOR.</th>
<th>.EQV.</th>
<th>.NEQV.</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>OPERAND</th>
<th>RESULT of Unary Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>* BYTE</td>
<td>INTEGER(1) *</td>
</tr>
<tr>
<td>LOGICAL(1)</td>
<td>LOGICAL(1)</td>
</tr>
<tr>
<td>LOGICAL(2)</td>
<td>LOGICAL(2)</td>
</tr>
<tr>
<td>LOGICAL(4)</td>
<td>LOGICAL(4)</td>
</tr>
<tr>
<td>LOGICAL(8)</td>
<td>LOGICAL(8)</td>
</tr>
<tr>
<td>* Typeless</td>
<td>Default integer *</td>
</tr>
</tbody>
</table>

**Note:** * Resultant types for unitary operations in XL Fortran

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

---

**IBM Extension**

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 12. Result Types for binary logical expressions**

<table>
<thead>
<tr>
<th>first operand</th>
<th>second operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>*BYTE</td>
<td>*BYTE</td>
</tr>
<tr>
<td>*BYTE</td>
<td>LOGICAL(1)</td>
</tr>
<tr>
<td>LOGICAL(1)</td>
<td>LOGICAL(1)</td>
</tr>
<tr>
<td>LOGICAL(2)</td>
<td>LOGICAL(2)</td>
</tr>
<tr>
<td>LOGICAL(4)</td>
<td>LOGICAL(4)</td>
</tr>
<tr>
<td>LOGICAL(8)</td>
<td>LOGICAL(8)</td>
</tr>
<tr>
<td>*Typeless</td>
<td>*Typeless</td>
</tr>
<tr>
<td>*Typeless</td>
<td>LOGICAL(1)</td>
</tr>
<tr>
<td>*Typeless</td>
<td>LOGICAL(2)</td>
</tr>
<tr>
<td>*Typeless</td>
<td>LOGICAL(4)</td>
</tr>
<tr>
<td>*Typeless</td>
<td>LOGICAL(8)</td>
</tr>
<tr>
<td></td>
<td>*Typeless</td>
</tr>
</tbody>
</table>

**Note:** * Resultant types for binary logical expressions in XL Fortran

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] primary
```
**defined_unary_op**

is a defined unary operator. See “Extended intrinsic and defined operations” on page 96.

**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:

<table>
<thead>
<tr>
<th>Relational Operator</th>
<th>Representing</th>
</tr>
</thead>
<tbody>
<tr>
<td>.LT. or &lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>.LE. or &lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>.EQ. or ==</td>
<td>Equal to</td>
</tr>
<tr>
<td>.NE. or *&lt;&gt; or /=</td>
<td>Not equal to</td>
</tr>
<tr>
<td>.GT. or &gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>.GE. or &gt;=</td>
<td>Greater than or equal to</td>
</tr>
</tbody>
</table>

**Note:** * XL Fortran relational operator.

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:**

```fortran
  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 and char_expr2 are each character expressions
relational_operator is any of the relational operators described in "Arithmetic relational expressions" on page 95.

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 LLL) 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.

IBM Extension

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

End of IBM Extension

Example of a character relational expression:

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

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 (see "Interface blocks" on page 141). 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 144 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:

\[ L \text{ .OR. } (A + B) \text{ .GE. } C \]

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

\[ L \text{ .OR. } ((A + B) \text{ .GE. } C) \]

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 81. 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:

\[
\begin{align*}
A(I) & = \text{FUNC1}(I) \\
Y & = \text{FUNC2}(X) + X
\end{align*}
\]

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.

---

IBM Extension

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 (+, -):

<table>
<thead>
<tr>
<th>Data Type of OPERAND</th>
<th>Data Type of RESULT of Unary Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE</td>
<td>INTEGER(1)</td>
</tr>
<tr>
<td>INTEGER(n)</td>
<td>INTEGER(n)</td>
</tr>
<tr>
<td>LOGICAL(n)</td>
<td>LOGICAL(n)</td>
</tr>
<tr>
<td>Typeless</td>
<td>Default integer</td>
</tr>
</tbody>
</table>

where \( n \) represents the kind type parameter. \( n \) must not be replaced with a logical constant even if \(-qintlog\) is on, nor by a character constant even if \(-qctyplss\) is on, nor can it be a typeless constant. 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:

<table>
<thead>
<tr>
<th>first operand</th>
<th>second operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE</td>
<td>INTEGER(1)</td>
</tr>
<tr>
<td>INTEGER(x)</td>
<td>INTEGER(x)</td>
</tr>
<tr>
<td>LOGICAL(x)</td>
<td>INTEGER(x)</td>
</tr>
<tr>
<td>Typeless</td>
<td>INTEGER(1)</td>
</tr>
</tbody>
</table>

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 [XL Fortran Compiler Reference](#) for information about compiler options.

| End of IBM Extension |
Using BYTE data objects

IBM Extension

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 (VALUE attribute), 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.

End of IBM Extension

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. See "Defined assignment" on page 147.

The general form of an intrinsic assignment is:
The shapes of `variable` and `expression` must conform. `variable` must be an array if `expression` is an array (see “Expressions involving arrays” on page 81). 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 76 for details), and neither `variable` nor `expression` can be an assumed-size array. The types of `variable` and `expression` must conform as follows:

<table>
<thead>
<tr>
<th>Type of variable</th>
<th>Type of expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric</td>
<td>Numeric</td>
</tr>
<tr>
<td>Logical</td>
<td>Logical</td>
</tr>
<tr>
<td>Character</td>
<td>Character</td>
</tr>
<tr>
<td>Derived type</td>
<td>Derived type (same as <code>variable</code>)</td>
</tr>
</tbody>
</table>

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 declared type as the variable. (See “Determining declared type for derived types” on page 33 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. For an allocatable component the following sequence of operations is applied:

1. If the component of `variable` is currently allocated, it is deallocated.
2. If the component of expression is currently allocated, the corresponding component of variable is allocated with the same type and type parameters as the component of expression. If it is an array, it is allocated with the same bounds.

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:

<table>
<thead>
<tr>
<th>Type of variable</th>
<th>Value Assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>( \text{INT(expression,KIND=KIND(variable))} )</td>
</tr>
<tr>
<td>Real</td>
<td>( \text{REAL(expression,KIND=KIND(variable))} )</td>
</tr>
<tr>
<td>Complex</td>
<td>( \text{CMPLX(expression,KIND=KIND(variable))} )</td>
</tr>
</tbody>
</table>

**IBM Extension**

Note: Arithmetic integer operations for INTEGER(8) data items, including intermediate results, are performed using INTEGER(8) arithmetic in both 32-bit and 64-bit mode. Arithmetic integer operations for INTEGER(1), INTEGER(2), and INTEGER(4) data objects, including intermediate results, are performed using INTEGER(4) arithmetic in 32-bit mode and INTEGER(8) arithmetic in 64-bit mode. If an intermediate result is used in a context requiring a smaller integer size, it is converted as required.

**End of IBM Extension**

**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.

**End of IBM Extension**

**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 = 'ABCDEFGH'I'

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 'ACDEEFGH'I'

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
  where_body_construct
  masked_ELSEWHERE_block
  ELSEWHERE_block
  END_WHERE_statement

WHERE_construct_statement
See "WHERE" on page 423 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 290 for syntax details.

ELSEWHERE_block

ELSEWHERE_statement
is an ELSEWHERE statement that does not specify a mask_expr. See "ELSEWHERE" on page 290 for syntax details.

END_WHERE_statement
See "END (Construct)" on page 293 for syntax details.

Rules:
- mask_expr is a logical array expression.
- In each where_assignment_statement, the mask_expr and the variable being defined must be arrays of the same shape.
- 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.
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
  - 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. 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 $m_c \cdot \text{AND. mask_expr}$.
    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 \( m_c . \AND. mask\_expr \).

**End of Fortran 95**

- Otherwise:
  1. The compiler evaluates the \( mask\_expr \), and assigns \( m_c \) the value of that \( mask\_expr \).
  2. \( m_p \) becomes \( . \NOT. mask\_expr \).

**End of Fortran 95**

**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 \( m_c . \AND. mask\_expr \).

**End of Fortran 95**

**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):

```fortran
WHERE (mask1) ! W1 *
  WHERE (mask2) ! W2 *
    ... ! W3 *
    ELSEWHERE (mask3) ! W4 *
    ... ! W5 *
    END WHERE ! W6 *
    ELSEWHERE (mask4) ! W7 *
      ... ! W8 *
      ELSEWHERE ! W9
      ... ! W10
      END WHERE ! W11
```

**Note:** * Fortran 95

The compiler sets control and pending control masks as it executes each statement, as shown below:

```
<table>
<thead>
<tr>
<th>Statement</th>
<th>Fortran 95</th>
</tr>
</thead>
</table>
| W1        | $m_c = \text{mask1}$  
|           | $m_p = \text{.NOT. mask1}$ |
| W2        | $m_p = \text{mask1 .AND. (.NOT. mask2)}$  
|           | $m_c = \text{mask1 .AND. mask2}$ |
| W4        | $m_c = \text{mask1 .AND. (.NOT. mask2)}$  
|           | $m_p = \text{mask1 .AND. (.NOT. mask2) .AND. (.NOT. mask3)}$  
|           | $m_c = \text{mask1 .AND. (.NOT. mask2) .AND. mask3}$ |
| W6        | $m_c = \text{mask1}$  
|           | $m_p = \text{.NOT. mask1}$ |
```

End of Fortran 95

```
<table>
<thead>
<tr>
<th>Statement</th>
<th></th>
</tr>
</thead>
</table>
| W7        | $m_c = \text{.NOT. mask1}$  
|           | $m_p = \text{( .NOT. mask1) .AND. ( .NOT. mask4)}$  
|           | $m_c = \text{(.NOT. mask1) .AND. mask4}$ |
```
Statement W9
    \( m_c = (\text{.NOT.} \, \text{mask1}) \, \text{.AND.} \, (\text{.NOT.} \, \text{mask4}) \)
Statement W11
    \( m_c = 0 \)
    \( m_p = 0 \)

The compiler uses the values of the control masks set by statements W2, W4, W7, and W9 when it executes the respective where_assignment_statements 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) \quad \text{! Only the positive elements of A are used in the LOG calculation.}
  B = A \quad \text{! The mask uses the original array A instead of the new array A.}
  C = A / SUM(LOG(A)) \quad \text{! 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) \quad \text{! 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.

--- End of Fortran 95 ---

--- IBM Extension ---

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 444.

--- End of IBM Extension ---

--- Fortran 95 ---

FORALL_construct_statement

See "FORALL (construct)" on page 314 for syntax details.

END_FORALL_statement

See "END (Construct)" on page 293 for syntax details.

forall_body

is one or more of the following statements or constructs:

forall_assignment

WHERE statement (see "WHERE" on page 423)
WHERE construct (see "WHERE construct" on page 103)
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.

End of Fortran 95

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:

   ```fortran
   FORALL (I=1:3, J=4:5)
   ```

   The set of combinations of I and J is:
   ```fortran
   {(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:

   ```fortran
   {(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.

   ```fortran
   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
   END FORALL
   ```

   **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 105. 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.

```fortran
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
    ELSEWHERE  ! greater than 0.
    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.

```fortran
! Same as FORALL (I=1:100,J=1:100,I.NE.J) A(I,J)=A(J,I)
```

```fortran
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. If it is a variable, it must have the `TARGET` attribute (or be a subobject of such an object) or the `POINTER` attribute. If it is an expression, it must yield a value that has the `POINTER` attribute.
A pointer object must have 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.

--- IBM Extension ---

A pointer of type byte can only be associated with a target of type byte, INTEGER(1), or LOGICAL(1).

--- End of IBM Extension ---

Any previous association between a 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. If target of a pointer assignment is an allocatable object, it must be allocated.

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 240 for an alternative form of associating a pointer with a target.

**Examples of pointer assignment**

```fortran
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
```

Chapter 5. Expressions and assignment 113
T_VAR = T(P)        ! T_VAR%COMP_PTR points to MYVAR
ARR => DARG(1:3)
BYTE_PTR => LOG_PTR
END -

Procedure pointer assignment

The procedure pointer assignment statement causes a procedure pointer to become
associated with a target or causes the procedure pointer’s association status to
become disassociated or undefined.

```
proc_pointer_object => proc_target
```

proc_target
is an expression or a procedure name. If proc_target is an expression, it
must be a function that returns a procedure pointer. If proc_target is a
procedure name must be the name of an external procedure, module
procedure, dummy procedure, or another procedure pointer. proc_target
must not be an elemental procedure.

proc_pointer_object
is a procedure pointer.

If proc_target is not a procedure pointer, proc_pointer_object becomes associated with
proc_target. If proc_target is a procedure pointer and is associated with a procedure,
proc_pointer_object becomes associated with the same procedure. If proc_target is a
pointer with an association status of disassociated or undefined, proc_pointer_object
acquires the same status.

If the proc_pointer_object has an explicit interface, its characteristics must be the
same as proc_target except that proc-target can be pure even if proc_pointer_object is
not. If the characteristics of proc_pointer_object or proc_target are such that an explicit
interface is required, both proc_pointer_object and proc-target must have an explicit
interface.

If proc_pointer_object has an implicit interface and is explicitly typed or referenced
as a function, proc-target must be a function. If proc_pointer_object has an implicit
interface and is referenced as a subroutine, proc-target must be a subroutine.

If proc-target and proc_pointer_object are functions, they must have the same type;
corresponding type parameters must either be both deferred or have the same
value.

If proc-target is a specific procedure name that is also a generic name, only the
specific procedure is associated with proc_pointer_object.

Related information::
* "PROCEDURE" on page 371

End of Fortran 2003 Standard
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) ! Prints '1'
PRINT *, INT_TEMPLATE ! Add 4 to reach next element
P = P + 4; ! 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
```

End of IBM Extension
Chapter 6. Execution Control

You can control the execution of a program sequence using constructs. Constructs contain statement blocks and other executable statements that can alter the normal execution sequence. This section contains detailed descriptions of the following constructs:

- ASSOCIATE
- DO
- DO WHILE
- IF
- SELECT CASE

Detailed syntax diagrams for the constructs in this section can be found by following the links to the associated statements.

For nesting to occur, a construct must be wholly contained within another construct. If a statement specifies a construct name, it applies to that construct. If the statement does not specify a construct name, the statement applies to the innermost construct in which it appears.

In addition to constructs, XL Fortran provides branching as a method for transferring control from one statement to another statement in the same scoping unit.

**statement blocks**

A statement block consists of a sequence of zero or more executable statements, executable constructs, FORMAT statements, or DATA statements embedded in another executable construct and are treated as a single unit.

Within a program, you can not transfer control from outside of the statement block to within the statement block. You can transfer control within the statement block, or from within the statement block to outside the block. For example, you can have a GO TO statement branching to a label that is within a statement block. You can not branch from a GO TO statement outside the statement block.

**ASSOCIATE Construct**

The ASSOCIATE construct creates an association between an identifier and a variable, or the value of an expression, during the execution of that construct. The identifier you specify in an ASSOCIATE construct becomes an associating entity. You can create multiple associating entities inside a single ASSOCIATE construct.
Syntax

\[
\begin{align*}
\text{ASSOCIATE statement} & \quad \text{See "ASSOCIATE" on page 243 for syntax details} \\
\text{ASSOCIATE statement block} & \quad \text{See "END (Construct)" on page 293 for syntax details} \\
\text{END ASSOCIATE statement} & \quad \text{See "END (Construct)" on page 293 for syntax details}
\end{align*}
\]

ASSOCIATE_statement

Execution of an ASSOCIATE construct causes execution of an ASSOCIATE_statement followed by the ASSOCIATE_statement_block. During execution of that block, the construct creates an association with an identifier and the corresponding selector. The associating entity assumes the declared type and type parameters of the selector.

Examples

The following example uses the ASSOCIATE construct as a shorthand for a complex expression and renames an existing variable, MYREAL. After the end of the ASSOCIATE construct, any change within the construct to the value of the associating entity that associates with MYREAL is reflected.

```fortran
PROGRAM ASSOCIATE_EXAMPLE

REAL :: MYREAL, X, Y, THETA, A
X = 0.42
Y = 0.35
MYREAL = 9.1
THETA = 1.5
A = 0.4

ASSOCIATE ( Z => EXP(-(X**2+Y**2)) * COS(THETA), V => MYREAL)
    PRINT *, A+Z, A-Z, V
    V = V * 4.6
END ASSOCIATE

PRINT *, MYREAL

END PROGRAM ASSOCIATE_EXAMPLE
```

The expected output is.

0.4524610937 0.3475389183 9.100000381
41.86000061

End of Fortran 2003 Standard
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.

The **DO statement**

![DO_statement]

- **DO_statement**  See “**DO**” on page 280 for syntax details
- **statement_block**
- **END_DO_statement**

- **terminal_statement**  See “**END (Construct)**” on page 293 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 and attributes,” on page 235 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 compatible with the restrictions on a logical **IF** 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.

A labeled **DO** statement must be terminated with an **END DO** statement that has a matching statement label. A **DO** statement with no label must be terminated with an unlabeled **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 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 does not have an iteration count limit or a termination condition.

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 102.) 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:

\[
\text{MAX} \left( \text{INT} \left( \frac{(m_2 - m_1 + m_3)}{m_3} \right), 0 \right)
\]

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.

IBM Extension

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.

End of IBM Extension
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.

**DO execution range**
The range of a **DO** construct includes all statements within the statement block. These statements execute until reaching the terminal statement. A **DO** variable must not become redefined or undefined during execution of the range of a **DO** construct, and only becomes redefined through incremental processing.

**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 90/95/2003 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:

```fortran
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
  statement_block
  END_DO_statement
```

- **DO WHILE_statement**
  
  See “**DO WHILE**” on page 282 for syntax details

- **END_DO_statement**
  
  See “**END (Construct)**” on page 293 for syntax details

- **terminal_stmt**
  
  is a statement that terminates the **DO WHILE** construct. See “**The terminal statement**” on page 119 for details.

The rules applicable to the **DO construct** names and ranges, active and inactive **DO** constructs, and terminal statements also apply to the **DO WHILE** construct.

Example

```fortran
I=10
TWO_DIGIT: DO WHILE ((I.GE.10).AND.(I.LE.99))
  J=J+1
  READ (5,+) I
END DO TWO_DIGIT
END
```
**IF construct**

The IF construct selects no more than one of its statement blocks for execution.

```
>>> Block_IF_statement

>>> statement_block

>>> ELSE_IF_block

>>> ELSE_block

>>> END_IF_statement

Block_IF_statement
See "IF (block)" on page 325 for syntax details.

END_IF_statement
See "END (Construct)" on page 293 for syntax details.

ELSE_IF_block

```

```
>>> ELSE_IF_statement

>>> statement_block

ELSE_IF_statement
See "ELSE IF" on page 290 for syntax details.

ELSE_block

```

```
>>> ELSE_statement

>>> statement_block

ELSE_statement
See "ELSE" on page 289 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

```plaintext
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
```

**SELECT 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

CASE_statement_block

END_SELECT_statement
```

**SELECT_CASE_statement** defines the case expression that is to be evaluated. See “SELECT CASE” on page 395 for syntax details.
END_SELECT_statement
terminates the CASE construct. See "END (Construct)" on page 293 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 254 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 254.)
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.
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

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 321 for syntax details.
- **Computed GO TO**
  transfers control to possibly one of several executable statements. See “GO TO (computed)” on page 322 for syntax details.
- **Unconditional GO TO**
  transfers control to a specified executable statement. See “GO TO (unconditional)” on page 323 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 324 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-of-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 section describes:
- "Scope"
- "Association" on page 134
- "Program units, procedures, and subprograms" on page 138
- "Interface blocks" on page 141
- "Generic interface blocks" on page 144
- "Main program" on page 148
- "Modules" on page 149
- "Block data program unit" on page 153
- "Function and subroutine subprograms" on page 154
- "Intrinsic procedures" on page 156
- "Arguments" on page 157
- "Argument association" on page 160
- "Recursion" on page 171
- "Pure procedures" on page 171
- "Elemental Procedures" on page 174

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. Using the `IMPORT` statement, an interface body can also access names from its host.
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

---

**IBM Extension**

- CRITICAL lock_names

---

**End of IBM Extension**

---

**Fortran 2003 Standard**

- Entities that have binding labels.

---

**End of Fortran 2003 Standard**

If a name identifies a global entity, it cannot be the same as any binding label in the same executable program, and it cannot be used to identify any other global entity in the same executable program unless that entity is an intrinsic module.
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. Components of a record structure are local entities of class 2. A separate class exists for each type.

A name declared to be a derived type may have the same name as another local entity of class 1 of that scoping unit that is not a derived type. In this case, the structure constructor for that type is not available in that scope. Similarly, a local entity of class 1 is accessible via host association or use association, even if there is another local entity of class 1 accessible in that scope, if

- one of the two entities is a derived type and the other is not; and
- in the case of host association, the derived type is accessible via host association.

For example, given a module M, a program unit P, and an internal subprogram or module subprogram S nested in P, if you have an entity named T1 declared in M that is accessed by use association in P (or in S), you can declare another entity in P (or in S, respectively) with the same name T1, so long as one of the two is a derived type. If you have an entity named T2 accessible in P, and an entity named T2 declared in S, then the T2 accessible in P is accessible in S if the T2 in P is a derived type. If the T2 in P was not a derived type, it would not be accessible in S if S declared another T2 (of derived type or not).
The structure constructor for that type will not be available in that scope. A local entity of class 1 in a scope that has the same name as a derived type accessible in that scope must be explicitly declared in a declaration statement in that scope.

If two local entities of class 1, one of which is a derived type, are accessible in a scoping unit, any PUBLIC or PRIVATE statement that specifies the name of the entities applies to both entities. If the name of the entities is specified in a VOLATILE statement, the entity or entities declared in that scope have the volatile attribute. If the two entities are public entities of a module, any rename on a USE statement that references the module and specifies the names of the entities as the use_name applies to both entities.

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 169 for details.

**Statement and construct entities**

**Statement entities:** The following items are statement 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.

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.

**Statement and construct entity:**

<table>
<thead>
<tr>
<th>Fortran 95</th>
</tr>
</thead>
<tbody>
<tr>
<td>The following is a statement and/or construct entity:</td>
</tr>
<tr>
<td>• Name of a variable that appears as an index_name in a FORALL statement or FORALL construct.</td>
</tr>
<tr>
<td>– SCOPE: Scope of the FORALL statement or construct.</td>
</tr>
</tbody>
</table>

The only attributes held by the FORALL statement or construct entity 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.

| End of Fortran 95 |

**Construct entity:**

<table>
<thead>
<tr>
<th>Fortran 2003 Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>The following is a construct entity:</td>
</tr>
<tr>
<td>• The associate name of an ASSOCIATE construct</td>
</tr>
<tr>
<td>– SCOPE: Scope of the block of the ASSOCIATE construct.</td>
</tr>
</tbody>
</table>

If the name of a global or local entity accessible in the scoping unit of an ASSOCIATE construct is the same as an associate name, the name is interpreted within the block of the ASSOCIATE construct as that of the associate name. Elsewhere in the scoping unit, the name is interpreted as the global and local entities.

| End of Fortran 2003 Standard |
Association

Association exists if the same data can be identified with different names in the same scoping unit, or if the same data can be accessed in different scoping units of the same executable program. (See “Argument association” on page 160 for information on argument association in procedures and functions.)

Host association

Host association allows an internal subprogram, module subprogram, interface body, or derived-type definition to access named entities that exist in its host. In interface bodies, entities cannot be accessed by host association unless they are made accessible by an IMPORT statement. Accessed entities have the same attributes and are known by the same name as they are in the host.

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
- A name of a derived type
- An object name in a type declaration, EQUIVALENCE, POINTER, ALLOCATABLE, 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
- The name of an entity declared by an interface body or PROCEDURE declaration statement

Entities in the host scoping unit that have the same name as a local entity are not accessible by host association.

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. For more information on events that cause definition and undefinition of variables, see “Definition status of variables” on page 52.

The host scoping unit of an internal or module subprogram can contain the same use-associated entities.

**Example of host association**

```fortran
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 414 for details.

```fortran
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
```

**Construct Association**

Construct association establishes an association between each selector and the corresponding associate name of the construct. Each associate name remains associated with the corresponding selector throughout the execution of the executed block. Within the block, each selector is known by and may be accessed by the corresponding associate name. Upon termination of the construct, the association is terminated.
**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.
  
  ```fortran
  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.
  
  ```fortran
  P => T
  ```

**Disassociated**

- The pointer is nullified by a **NULLIFY** statement or by the `qinit=f90ptr` option. See `qinit` in the **XL Fortran Compiler Reference**.
  
  ```fortran
  NULLIFY (P)
  ```

---

**End of Fortran 2003 Standard**

- The pointer is successfully deallocated.
  
  ```fortran
  DEALLOCATE (P)
  ```

- The pointer is pointer-assigned to a disassociated pointer.
  
  ```fortran
  NULLIFY (Q); P => Q
  ```

**Undefined**

- Initially (unless the `qinit=f90ptr` option is specified)
  
  ```fortran
  ```

---

**End of Fortran 2003 Standard**

- The pointer is an ultimate component of an object with default initialization specified for the component and:
  
  - a procedure is invoked with this object as an actual argument corresponding to a nonpointer, nonallocatable dummy argument with **INTENT(OUT)**,
  
  - a procedure with the object as an unsaved nonpointer, nonallocatable local object that is not accessed by use or host association is invoked,
  
  - this object is allocated, or

---

**End of Fortran 2003 Standard**

- If it is pointer-assigned to a pointer whose association status is undefined.

- 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

IBM Extension

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.

End of IBM Extension
Program units, procedures, and subprograms

A program unit is a sequence of one or more lines, organized as statements, comments, and 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.

Main programs, external procedures, block data program units, common blocks, entities with binding labels, and modules are global entities. 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

<table>
<thead>
<tr>
<th>Fortran 2003 Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>• The BIND attribute is not allowed on an internal procedure</td>
</tr>
</tbody>
</table>

End of Fortran 2003 Standard
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

To ascertain the characteristics of a procedure:

• 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, allocatable, a pointer, a target, or has the \texttt{VALUE} attribute. 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, it specifies the characteristics of the result value, specifically:
  – Type
  – Any type parameters
  – Rank
  – Whether the result value is a pointer

Migration Tip:

Turn your external procedures into internal subprograms or put them into modules. The explicit interface provides type checking.

\textbf{FORTRAN 77 source}

\begin{verbatim}
PROGRAM MAIN
  INTEGER A
  A=58
  CALL SUB(A)  ! C must be passed
END
SUBROUTINE SUB(A)
  INTEGER A,B,C
  C=A+B  ! A must be redeclared
END
\end{verbatim}

\textbf{Fortran 90/95/2003 source:}

\begin{verbatim}
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
\end{verbatim}

\textbf{Chapter 7. Program units and procedures 139}
– Whether the result value is allocatable.

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.

• Determine whether the procedures is \texttt{PURE} or \texttt{ELEMENTAL}.
• Determine whether procedure has the \texttt{BIND} attribute.

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:

<table>
<thead>
<tr>
<th>Entity</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dummy procedure</td>
<td>Explicit in a scoping unit if an interface block exists or is accessible, or if an explicit interface is specified by a \texttt{PROCEDURE} declaration statement. Implicit in all other cases.</td>
</tr>
<tr>
<td>External subprogram</td>
<td>Explicit in a scoping unit other than its own if an interface block exists or is accessible, or if an explicit interface is specified by a \texttt{PROCEDURE} declaration statement. Implicit in all other cases.</td>
</tr>
<tr>
<td>Recursive procedure with a result clause</td>
<td>Explicit in the subprogram’s own scoping unit.</td>
</tr>
<tr>
<td>Module procedure</td>
<td>Always explicit.</td>
</tr>
<tr>
<td>Internal procedure</td>
<td>Always explicit.</td>
</tr>
<tr>
<td>Generic procedure</td>
<td>Always explicit.</td>
</tr>
<tr>
<td>Intrinsic procedure</td>
<td>Always explicit.</td>
</tr>
<tr>
<td>Statement function</td>
<td>Always implicit.</td>
</tr>
</tbody>
</table>

Internal subprograms cannot appear in an interface block or in a \texttt{PROCEDURE} declaration statement.

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 or in a \texttt{PROCEDURE} declaration statement.

**Explicit interface**

A procedure must have an explicit interface in any of the following cases:

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
   • \texttt{f95} in a context that requires it to be pure \texttt{f95}

2. The procedure has
   • a dummy argument that has the \texttt{ALLOCATABLE} or \texttt{OPTIONAL}, \texttt{POINTER}, \texttt{TARGET} or \texttt{VALUE} attributes.
   • an array-valued result (for functions only)
   • a result whose length type parameter is neither assumed nor constant (for character functions only)
• a pointer or allocatable result (for functions only)
• a dummy argument that is an assumed-shape array

3. [F95] The procedure is elemental. [F95]
4. The procedure has the [F2003] BIND attribute.

**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. The interface for a procedure can also be specified by a [PROCEDURE] declaration statement.

```
 interface
  function_interface_body
  subroutine_interface_body
  module_procedure_statement
end_interface
```

**INTERFACE_statement**

See "INTERFACE" on page 343 for syntax details

**END_INTERFACE_statement**

See "END INTERFACE" on page 296 for syntax details

**MODULE_PROCEDURE_statement**

See "MODULE PROCEDURE" on page 352 for syntax details

**FUNCTION_interface_body**

```
 function
  specification_part
end_function
```

See "FUNCTION" on page 344 for syntax details
SUBROUTINE_interface_body

FUNCTION_statement, SUBROUTINE_statement
For syntax details, see “FUNCTION” on page 317 and “SUBROUTINE” on page 401.

specification_part
is a sequence of statements from the statement groups numbered 2, 3 and 5 in “Order of statements and execution sequence” on page 14.

end_function_statement, end_subroutine_statement
For syntax details of both statements, see “END” on page 292.

In an interface body or with a procedure declaration statement, you specify all the characteristics of the procedure. See “Interface concepts” on page 139. 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 unless you specify the IMPORT statement. It is treated as if it had a host with the default implicit rules. See “How type is determined” on page 51 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 343 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 in any accessible interface block with the same generic identifier.

---

**IBM Extension**

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 159 for details.

---

### Example of an interface Block

```fortran
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(I)
! 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

In an INTERFACE statement, a generic interface block must specify one of the following:
- a generic name
- defined operator
- defined assignment

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.

<table>
<thead>
<tr>
<th>Fortran 2003 Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Within a scoping unit, two procedures that have the same generic name must both be subroutines or both be functions.</td>
</tr>
<tr>
<td>End of Fortran 2003 Standard</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>IBM Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notes:</td>
</tr>
<tr>
<td>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.</td>
</tr>
<tr>
<td>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.</td>
</tr>
<tr>
<td>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.</td>
</tr>
<tr>
<td>End of IBM Extension</td>
</tr>
</tbody>
</table>
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)
    ! Here, EXT is a variable
    INTEGER I
    REAL EXT
  END FUNCTION FUNC2
END INTERFACE
EXTERNAL MYFUNC
IRESULT = A(INTVAL)  ! Call to function AI
RESULT = 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
    FUNCTION MAXCHAR(STRING)
      CHARACTER(50) STRING
    END FUNCTION MAXCHAR
  END INTERFACE
  INTERFACE ABS
    FUNCTION MYABS(ARG)
      REAL(8) MYABS, ARG
    END FUNCTION MYABS
  END INTERFACE
  REAL(8) DARG, DANS
  REAL(4) RANS
  INTEGER IANS, IARG
  CHARACTER(50) NAME
  DANS = ABS(DARG)  ! Calls external MYABS
```

Chapter 7. Program units and procedures 145
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 96). 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, $d_1$.

   Fortran 2003 Standard

   b. 1) the $generic_{spec}$ in an INTERFACE statement specifies OPERATOR ($op$), or

   End of Fortran 2003 Standard

   c. The type of $x_1$ is the type of the dummy argument $d_1$.
   d. The type parameters, if any, of $x_1$ must match those of $d_1$.
   e. Either
      • The function is ELEMENTAL, or
      • The rank of $x_1$, and its shape, if it is an array, match those 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 $d_2$.

   Fortran 2003 Standard

   b. 1) the $generic_{spec}$ in an INTERFACE block specifies OPERATOR ($op$), or

   End of Fortran 2003 Standard

   c. The types of $x_1$ and $x_2$ are the types 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. Either:
      • The function is ELEMENTAL and $x_1$ and $x_2$ are conformable or,
      • 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. If the operator specified is an intrinsic operator, the number of function arguments must be consistent with the intrinsic uses of that operator.

8. 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.

IBM Extension

9. 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.

End of IBM Extension

Example of a defined operator

```fortran
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. Either:
      1) the generic_spec of an interface block specifies ASSIGNMENT (=), or
   c. The type parameters, if any, of \( x_1 \) and \( x_2 \) match those of \( d_1 \) and \( d_2 \),
      respectively.
   d. Either:
      • The subroutine is ELEMENTAL and either \( x_1 \) and \( x_2 \) have the same
         shape, \( x_2 \) is scalar, or
      • 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.

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.

--- IBM Extension ---

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.

--- End of IBM Extension ---

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**

```fortran
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

specification_part

execution_part

internal_subprogram_part

END_PROGRAM_statement
```

**PROGRAM_statement**

See “PROGRAM” on page 374 for syntax details
specification_part

is a sequence of statements from the statement groups numbered
2, 4, and 5 in “Order of statements and execution sequence”
on page 14

execution_part

is a sequence of statements from the statement groups numbered
4 and 6 in “Order of statements and execution sequence” on
page 14, and which must begin with a statement from statement
group 6

internal_subprogram_part

See “Internal procedures” on page 138 for details

END_PROGRAM_statement

See “END” on page 292 for syntax details

A main program cannot contain an ENTRY statement, nor can it specify an
automatic object.

IBM Extension

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.

End of IBM Extension

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 2003 Standard

There are two types of modules, intrinsic and nonintrinsic. XL Fortran provides
intrinsic modules, while nonintrinsic modules are user-defined.

An intrinsic module can have the same name as other global entities, such as
program units, common blocks, external procedures, critical sections, or binding
labels of global entities. A scoping unit must not access both an intrinsic module
and a non-intrinsic module with the same name.

End of Fortran 2003 Standard

IBM Extension

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
declare the global data as THREADPRIVATE or THREADLOCAL. See
“COMMON” on page 262, THREADLOCAL, and THREADPRIVATE in the XL
Fortran Optimization and Programming Guide for more information.

End of IBM Extension
MODULE_statement

See "MODULE" on page 351 for syntax details

specification_part

is a sequence of statements from the statement groups numbered 2, 4, and 5 in "Order of statements and execution sequence" on page 14.
module_subprogram_part:

CONTAINS_statement
See “CONTAINS” on page 271 for syntax details

END_MODULE_statement
See “END” on page 292 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.

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 414 for details.

IBM Extension

Integer pointers cannot appear in specification_part if the pointee specifies a dimension declarator with nonconstant bounds.

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 undefined” on page 55 for more information.

End of IBM Extension

A module is a host to any module procedures, interface blocks, 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:**

```fortran
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
```

**Fortran 90/95/2003 source:**

```fortran
MODULE FUNCS
    REAL A, B, C
    ! Common block no longer needed
    INTEGER NAME, NUMBER
    ! Global data
    CONTAINS
    SUBROUTINE CALLUP (PARM)
        ! Global data
        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**

```fortran
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
```
Block data program unit

A block data program unit provides initial values for objects in named common blocks.

```
Program Main
Use M            ! Main program accesses module M
Call Sub()       ! module M
End Program
```

A type declaration statement in a block data specification-part must not contain ALLOCATABLE or EXTERNAL attribute specifiers.

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

```fortran
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

! 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

specification_part

execution_part

internal_subprogram_part

end_subprogram_statement
```

subprogram_statement

See “FUNCTION” on page 317 or “SUBROUTINE” on page 401 for syntax details

specification_part

is a sequence of statements from the statement groups numbered 2, 4 and 5 in “Order of statements and execution sequence” on page 14

execution_part

is a sequence of statements from the statement groups numbered 4 and 6 in “Order of statements and execution sequence” on page 14 and which must begin with a statement from statement group 5

internal_subprogram_part

See “Internal procedures” on page 138 for details
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 any of the following:
  - execution of a CALL statement (see “CALL” on page 252 for details)
  - execution of a 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:

```
function_name(actual_argument_spec_list)
```

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.

IBM Extension

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 161.)
On entry to an allocatable function, the allocation status of the result variable becomes not currently allocated.

The function result variable may be allocated and deallocated any number of times during the execution of the function. However, it shall be currently allocated and have a defined value on exit from the function. Automatic deallocation of the result variable does not occur immediately on exit from the function, but instead occurs after execution of the statement in which the function reference occurs.

### Examples of subprograms and procedure references

```fortran
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()
    CUBE = A**3
END FUNCTION CUBE
END SUBROUTINE Q
```

### Examples of allocatable function results

```fortran
FUNCTION INQUIRE_FILES_OPEN() RESULT(OPENSENED_STATUS)
LOGICAL,ALLOCATABLE :: OPENED_STATUS(:)
INTEGER I,J
LOGICAL TEST
DO I=1000,0,-1
    INQUIRE(UNIT=I,OPENED=TEST,ERR=100)
    IF (TEST) EXIT
100 CONTINUE
END DO
ALLOCATE(OPENSENED_STATUS(0:I))
DO J=0,I
    INQUIRE(UNIT=J,OPENED=OPENED_STATUS(J))
END DO
END FUNCTION INQUIRE_FILES_OPEN
```

### Intrinsic procedures

An intrinsic procedure is a procedure already defined by XL Fortran. See Chapter 13, “Intrinsic procedures,” on page 473 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**

When you declare a data object 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 141. If the function already has the INTRINSIC attribute, it is extended; otherwise, it can be redefined.

---

### Arguments

#### Actual argument specification

<table>
<thead>
<tr>
<th>arg_keyword</th>
<th>argument</th>
<th>(1)</th>
<th>%VAL(---argument)</th>
<th>(2)</th>
<th>%REF(---argument)</th>
</tr>
</thead>
</table>

**Notes:**

1 IBM Extension

2 IBM Extension

arg_keyword is a dummy argument name in the explicit interface of the procedure being invoked

argument is an actual argument

---

Chapter 7. Program units and procedures 157
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 155. You cannot use a non-intrinsic elemental procedure as an actual argument in Fortran 95.

**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:**

```fortran
INTEGER MYARRAY(1:10)
INTERFACE
  SUBROUTINE SORT(ARRAY, DESCENDING, ARRAY_SIZE)
  SUBROUTINE OPTIONS
    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
```

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 155. You cannot use a non-intrinsic elemental procedure as an actual argument in Fortran 95.

**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:**

```fortran
INTEGER MYARRAY(1:10)
INTERFACE
  SUBROUTINE SORT(ARRAY, DESCENDING, ARRAY_SIZE)
  SUBROUTINE OPTIONS
    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
```
IF (PRESENT(DESCENDING)) THEN
  .
  .
END SUBROUTINE

**Dummy arguments**

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)

---

**Notes:**
1. IBM Extension
2. IBM Extension

%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 161 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.

If associated with a null argument in a procedure reference, the corresponding dummy argument is undefined.

IBM Extension

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, or the corresponding dummy argument has the VALUE attribute, the subprogram does not have access to the storage area of the actual argument.

End of IBM Extension

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.

<table>
<thead>
<tr>
<th>IBM Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>If you have programs that do not conform to these restrictions, using the compiler option <code>qalias=nostd</code> may be appropriate. See the <code>qalias Option</code> in the <code>XL Fortran Compiler Reference</code> for details.</td>
</tr>
</tbody>
</table>

End of IBM Extension

### %VAL and %REF

<table>
<thead>
<tr>
<th>IBM Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>To call subprograms written in languages other than Fortran (for example, user-written C programs, or Linux 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 <code>qnullterm</code> compiler option to ensure that scalar character initialization expressions are passed with terminating null strings. See <code>qnullterm</code> in the <code>XL Fortran Compiler Reference</code> for details.)</td>
</tr>
<tr>
<td>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.</td>
</tr>
<tr>
<td>%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.</td>
</tr>
<tr>
<td>The argument list built-in functions are:</td>
</tr>
<tr>
<td><strong>%VAL</strong> This built-in function can be used with actual arguments that are <code>CHARACTER(1)</code>, 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.</td>
</tr>
<tr>
<td>%VAL cannot be used with actual arguments that are arrays, procedure names, or character expressions of length greater than 1 byte.</td>
</tr>
</tbody>
</table>
| %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 passed on the left with zeros to a 32-bit value, regardless of whether the -qctypless 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
REAL XVAR
CALL RIGHT3(%VAL(XVAR)) ! real argument passed by value
IVARB=6
CALL TPROG(%VAL(IVARB)) ! integer argument passed by value
```

See "VALUE" on page 417 for a standard-conforming alternative to %VAL.

See Interlanguage calls in the XL Fortran Optimization and Programming Guide for more information.

End of IBM Extension

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 341 for syntax details.

IBM Extension

The following table outlines XL Fortran’s passing method for internal procedures (not including assumed-shape dummy arguments and pointer dummy arguments):

<table>
<thead>
<tr>
<th>Argument Type</th>
<th>Intent(IN)</th>
<th>Intent(OUT)</th>
<th>Intent(INOUT)</th>
<th>No Intent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-CHARACTER</td>
<td>VALUE</td>
<td>default</td>
<td>default</td>
<td>default</td>
</tr>
<tr>
<td>Scalar</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHARACTER*1</td>
<td>VALUE</td>
<td>REFERENCE</td>
<td>REFERENCE</td>
<td>REFERENCE</td>
</tr>
<tr>
<td>Scalar</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHARACTER*n</td>
<td>REFERENCE</td>
<td>REFERENCE</td>
<td>REFERENCE</td>
<td>REFERENCE</td>
</tr>
<tr>
<td>Scalar</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Derived Type</td>
<td>default</td>
<td>default</td>
<td>default</td>
<td>default</td>
</tr>
<tr>
<td>(*) Scalar</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Derived Type</td>
<td>VALUE</td>
<td>default</td>
<td>default</td>
<td>default</td>
</tr>
<tr>
<td>2 Scalar</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Derived Type</td>
<td>default</td>
<td>default</td>
<td>default</td>
<td>default</td>
</tr>
<tr>
<td>3 Scalar</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 158.
- 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 360 for details about syntax and rules.

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.

---

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.
• 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.

• If the optional dummy argument that is not present is allocatable, it must not be allocated, deallocated, or supplied as an actual argument corresponding to a nonallocatable dummy argument other than 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 16 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**

If the actual argument is scalar, the corresponding dummy argument must be scalar, unless the actual argument is an element or substring of an element of an array that is not an assumed-shape or pointer array. If the actual argument is allocatable, the corresponding dummy argument must also be allocatable. 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.

---

**Fortran 95**

The following apply to dummy arguments used in elemental subprograms:

• All dummy arguments must be scalar, and cannot have the ALLOCATABLE or 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 13, “Intrinsic procedures,” on page 473.

• A dummy argument cannot be an asterisk.

• A dummy argument cannot be a dummy procedure.

---

**End of Fortran 95**

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.

**Allocatable objects as dummy arguments**

An allocatable dummy argument has an actual argument which is also allocatable associated with it. If the allocatable dummy argument is an array, the associated actual argument must also be an array.

On procedure entry, the allocation status of an allocatable dummy argument becomes that of the associated actual argument. If the dummy argument is `INTENT(OUT)` and the associated actual argument is currently allocated, the actual argument is deallocated on procedure invocation so that the dummy argument has an allocation status of not currently allocated. If the dummy argument is not `INTENT(OUT)` and the actual argument is currently allocated, the value of the dummy argument is that of the associated actual argument.

While the procedure is active, an allocatable dummy argument that does not have `INTENT(IN)` may be allocated, deallocated, defined, or become undefined. No reference to the associated actual argument is permitted via another alias if any of these events occur.

On exit from the routine, the actual argument has the allocation status of the allocatable dummy argument (there is no change, of course, if the allocatable dummy argument has `INTENT(IN)`). The usual rules apply for propagation of the value from the dummy argument to the actual argument.

Automatic deallocation of the allocatable dummy argument does not occur as a result of execution of a `RETURN` or `END` statement in the procedure of which it is a dummy argument.

**Note:** An allocatable dummy argument that has the `INTENT(IN)` attribute must not have its allocation status altered within the called procedure. The main difference between such a dummy argument and a normal dummy argument is that it might be unallocated on entry (and throughout execution of the procedure).

**Example**

```fortran
SUBROUTINE LOAD(ARRAY, FILE)
   REAL, ALLOCATABLE, INTENT(OUT) :: ARRAY(:, :, :)
   CHARACTER(LEN=*), INTENT(IN) :: FILE
   INTEGER UNIT, N1, N2, N3
   INTEGER, EXTERNAL :: GET_LUN
   UNIT = GET_LUN() ! Returns an unused unit number
   OPEN(UNIT, FILE=FILE, FORM='UNFORMATTED')
   READ(UNIT) N1, N2, N3
   ALLOCATE(ARRAY(N1, N2, N3))
   READ(UNIT) ARRAY
   CLOSE(UNIT)
END SUBROUTINE LOAD
```

End of Fortran 2003 Standard
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.

IBM Extension

The passing method must be by reference; that is, %VAL must not be specified for the pointer actual argument.

End of IBM Extension

Procedures as dummy arguments

A dummy argument that is identified as a procedure or a procedure pointer is called a dummy procedure or dummy procedure pointer, respectively.

Fortran 2003 Standard

If a dummy argument is a dummy procedure without the POINTER attribute, the associated actual argument must be the specific name of an external procedure, module procedure, dummy procedure, or intrinsic procedure whose name can be passed as an argument, an associated procedure pointer, or a reference to a function that returns an associated procedure pointer. If the specific name is also a generic name, only the specific procedure is associated with the dummy argument.

If a dummy argument is a procedure pointer, the associated actual argument must be a procedure pointer, a reference to a function that returns a procedure pointer, or a reference to the NULL intrinsic function.

If an external procedure name or a dummy procedure name is used as an actual argument, its interface must be explicit or it must be explicitly declared with the EXTERNAL attribute.

If the interface of the dummy argument is explicit, the characteristics must be the same for the associated actual argument and the corresponding dummy argument, except that a pure actual argument may be associated with a dummy argument that is not pure.

If the interface of the dummy argument is implicit and either the name of the dummy argument is explicitly typed or it is referenced as a function, the dummy argument must not be referenced as a subroutine and the actual argument must be a function, function procedure pointer, or dummy procedure.

If the interface of the dummy argument is implicit and a reference to it appears as a subroutine reference, the actual argument must be a subroutine, subroutine...
procedure pointer, or dummy procedure.

Internal subprograms cannot be associated with a dummy procedure argument.

You cannot use a non-intrinsic elemental procedure as an actual argument in Fortran 95.

**Examples of procedures as dummy arguments**

```fortran
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
```

**Related information**

- See Chapter 13, “Intrinsic procedures,” on page 473 for details on which intrinsic procedures can be passed as actual arguments.
- See “Procedure references” on page 155 for the rules and restrictions for referencing a procedure.

**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**

```fortran
CALL SUB(*10)
STOP  ! STOP is never executed
10 PRINT *, 'RETURN 1'
CONTAINS
  SUBROUTINE SUB(*)
```

End of Fortran 2003 Standard


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 a non-elemental reference to 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 apply:

- 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:

```fortran
INTEGER N, RESULT
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 317, “SUBROUTINE” on page 401, or “ENTRY” on page 300.

IBM Extension

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.

End of IBM Extension

Pure procedures

Pure procedures are free of side effects and 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 be pure. 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(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] attribute. 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(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 must not be an asterisk (*) or refer to an external unit. The input/output statements are:

- [BACKSPACE]
- [CLOSE]
- [ENDIFILE]
- [F2003 FLUSH F2003]
- [INQUIRE]
- [OPEN]
- [PRINT]
- [READ]
- [REWIND]
• **WAIT**
• **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 be **INTENT(IN)**.
2. Subroutine dummy data objects with the **POINTER** attribute can 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**

```fortran
PROGRAM ADD
 INTEGER ARRAY(20,256)
 INTERFACE
   PURE FUNCTION PLUS_X(ARRAY) ! Interface required for
     INTEGER, INTENT(IN) :: ARRAY(:)
     INTEGER :: PLUS_X(SIZE(ARRAY))
 END FUNCTION
 END INTERFACE
 INTEGER :: X
 X = ABS(-4) ! Intrinsic function
 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 :: 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 :: Y
 Y=6
 ARRAY = ARRAY+Y
END SUBROUTINE
```

---

End of Fortran 95
Elemental Procedures

Fortran 95

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 must not have the `ALLOCATABLE` or `POINTER` attribute.
- The following apply to dummy arguments used in elemental subprograms:
  - All dummy arguments must be scalar, and must not have the `ALLOCATABLE` or `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 13, “Intrinsic procedures,” on page 473.
  - A dummy argument cannot be an asterisk.
  - A dummy argument cannot be a dummy procedure.
- Elemental subprograms must follow all of the rules that apply to pure subprograms, defined in “Pure procedures” on page 171.
- 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 88.

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 `IN`, `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 160].

Special rules apply to generic procedures that have an elemental specific procedure, see [“Resolving procedure references to generic names” on page 170].

Examples

Example 1:

! Example of an elemental function
PROGRAM P
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)

! The actual argument AA is a two-dimensional array. The procedure
! taking AA as an argument is not declared in the interface block.
! The specific procedure SCALAR_RAND is then called.
A = RAND(AA)

! The actual argument is a one-dimensional array section. The procedure
! taking a one-dimensional array as an argument is declared in the
! interface block. The specific procedure VECTOR_RANDOM is then called.
! This is a non-elemental reference since VECTOR_RANDOM is not elemental.
A(:,1) = RAND(AA(6:10,2))
END
Chapter 8. XL Fortran Input/Output

XL Fortran supports both synchronous and asynchronous input/output (I/O). Synchronous I/O halts an executing application until I/O operations complete. Asynchronous I/O allows an application to continue processing while I/O operations occur in the background. Both I/O types support the following file access methods:

- Sequential access
- Direct access
- Stream access

Each method of access offers benefits and limitations based on the I/O concepts of, Records, Files and Units.

This section also provides explanations of the IOSTAT= specifier codes that can result when using XL Fortran I/O statements.

Records

A record contains a sequence of characters or values. XL Fortran supports three record types:

- formatted
- unformatted
- endfile

Formatted records

A formatted record consists of a sequence of ASCII characters that can print in a readable format. Reading a formatted record converts the data values from readable characters into an internal representation. Writing a formatted record converts the data from the internal representation into characters.

Unformatted records

An unformatted record contains a sequence of values in an internal representation that can contain both character and noncharacter data. An unformatted record can also contain no data. Reading or writing an unformatted record does not convert any data the record contains from the internal representation.

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 an internal or external sequence of records or file storage units. You determine the file access method when connecting a file to a unit. You can access an external file using three methods:

- **Sequential access**
- **Direct access**
- **Stream access**

You can only access an internal file sequentially.

**Definition of an external file**

You must associate an external file with an I/O device such as a disk, or terminal. An external file exists for a program when a program creates that file, or the file is available to that program for reading and writing. Deleting an external file ends the existence of that file. An external file can exist and contain no records.

<table>
<thead>
<tr>
<th>IBM Extension</th>
</tr>
</thead>
</table>

To specify an external file by a file name, you must designate a valid operating system file name. Each file name can contain a maximum of 255 characters. If you specify a full path name, it can contain a maximum of 1023 characters.

| End of IBM Extension |

The preceding I/O statement determines the position of an external file. You can position an external file to:

- The initial point, which is the position immediately before the first record, or the first file storage unit.
- The terminal point, which is the position immediately after the last record, or the last file storage unit.
- The current record, when the file position is within a record. Otherwise, there is no current record.
- The preceding record, which is the record immediately before the current record. If there is no current record, the preceding record is the record immediately before the current file position. A preceding record does not exist when the file position is at its initial point or within the first record of the file.
- The next record, which is the record immediately after the current record. If there is no current record, the next record is the record immediately after the current position. The next record does not exist when the file position is at the terminal point or within the last record of the file.

An external file can also have indeterminate position after an error.

**File access methods**

**Sequential access**

Using sequential access, records in a file are read or written based on the logical order of records in that file. Sequential access supports both internal and external files.

**External files**: 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 \texttt{direct} \texttt{F2003} or stream access \texttt{F2003} I/O statements during the time the file is connected for sequential access.

**Internal files:** An internal file is a character variable that is not an array section with a vector subscript. You do not need to create internal files. They always exist, and are available to the application.

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.

An internal file must contain only formatted records \texttt{READ} and \texttt{WRITE} are the only statements that can specify an internal file. If a \texttt{WRITE} statement writes less than an entire record, blanks fill the remainder of that record.

**Direct access**

Using direct access, the records of an external file can be read or written in any order. The records must be either all formatted or all unformatted. The records must not be read or written using sequential or stream access, list-directed or namelist formatting, or a nonadvancing input/output statement. If the file was previously connected for sequential access, the last record of the file is an endfile record. The endfile record is not considered a part of the file connected for direct access.

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 \texttt{OPEN} statement when the file is connected.

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.

**Stream access**

\begin{center}
\textbf{Fortran 2003 Standard}
\end{center}

You can connect external files for stream access as either formatted or unformatted. Both forms use external stream files composed of one byte file storage units. While a file connected for unformatted stream access has only a stream structure, files connected for formatted stream access have both a record and a stream structure. These dual structure files have the following characteristics:

\begin{itemize}
  \item Some file storage units represent record markers.
  \item The record structure is inferred from the record markers stored in the file.
  \item There is no theoretical limit on record length.
  \item Writing an empty record without a record marker has no effect.
  \item If there is no record marker at the end of a file, the final record is incomplete but not empty.
\end{itemize}
• The endfile record in a file previously connected for sequential access is not considered part of the file when you connect that file for stream access.

The first file storage unit of a file connected for formatted stream access has a position of 1. The position of each subsequent storage unit is greater than the storage unit immediately before it. The positions of successive storage units are not always consecutive and positionable files need not be read or written to in order of position. To determine the position of a file storage unit connected for formatted stream access, use the POS= specifier of the INQUIRE statement. If the file can be positioned, you can use the value obtained using the INQUIRE statement to position that file. You read from the file while connected to the file, as long as the storage unit has been written to since file creation and that the connection permits a READ statement. File storage units of a file connected for formatted stream access can only be read or written by formatted stream access input/output statements.

The first file storage unit of a file connected for unformatted stream access has a position of 1. The position value of successive storage units is incrementally one greater than the storage unit it follows. Positionable files need not be read or written to in order of position. Any storage unit can be read from the file while connected to the file, if the storage unit has been written to since file creation and that the connection permits a READ statement. File storage units of a file connected for unformatted stream access can only be read or written by stream access input/output statements.

End of Fortran 2003 Standard

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 [UNIT=] for the form of a unit specifier.

Connection of a unit

A connection refers to the association between an external file and a unit. A connection must occur before the records of a 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

Preconnection

Preconnection occurs when the program begins executing. You can specify preconnection in I/O statements without the prior execution of an OPEN statement.

IBM Extension

Using formatted sequential access always preconnects units 0, 5 and 6 as unnamed files to the devices below:

• Unit 0 to the standard error device

• Unit 5 to the standard input device

• Unit 6 to the standard output device
The other properties of these files are the default specifier values for the `OPEN` statement with the following exceptions:

- `STATUS=’OLD’`
- `ACTION=’READWRITE’`
- `FORM=’FORMATTED’`

End of IBM Extension

**Implicit connection**

Implicit connection occurs when a sequential statement that is: `ENDFILE`, `PRINT`, `READ`, `REWIND`, or `WRITE` executes on a unit not already connected to an external file. The executing statement connects that unit to a file with a predetermined name. By default, this connection is unit n to file `fort.n`. You do not need to create the file before implicit connection. To implicitly connect to a different file name, see the `UNIT VARS` run-time option under Setting Run-Time Options in the XL Fortran Compiler Reference.

You can only connect a preconnected unit implicitly if you terminate the connection between the unit and the external file. In the next example a preconnected unit closes before implicit connection takes place.

**Sample Implicit Connection**

```fortran
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
```

A unit with an implicit connection uses the default specifier values of the `OPEN` statement, except for the `FORM=` and `ASYNCH=` specifiers. The first data transfer statement determines the values for `FORM=` and `ASYNCH=.`

If the first I/O statement uses format-directed, list-directed, or namelist formatting, the value of the `FORM=` specifier is set to `FORMATTED`. An unformatted I/O statement sets the specifier to `UNFORMATTED`.

If the first I/O statement is asynchronous, the value of the `ASYNCH=` specifier is set to `YES`. A synchronous I/O statement sets the specifier to `NO`.

End of IBM Extension

**Disconnection**

The `CLOSE` statement disconnects a file from a unit. You can connect the file again within the same program to the same unit or to a different unit. You can connect the unit again within the same program to the same file or a different file.

- You can not close unit 0
- You can not reconnect unit 5 to standard input after the unit closes
Data transfer statements

The **READ** statement obtains data from an **external** or **internal** file and transfers the data to internal storage. If you specify an input list, values transfer from the file to the data items you specify.

The **WRITE** statement transfers data from internal storage into an external or internal file.

The **PRINT** statement transfers data from internal storage into an external file. Specifying the **-qport=typestmt** compiler option enables the **TYPE** statement which supports functionality identical to **PRINT**. If you specify an output list and format specification, values transfer to the file from the data items you specify. If you do not specify an output list, the **PRINT** statement transfers a blank record to the output device unless the **FORMAT** statement it refers to contains, as the first specification, a character string edit descriptor or a slash edit descriptor. In this case, the records these specifications indicate transfer to the output device.

Execution of a **WRITE** or **PRINT** statement for a file that does not exist creates that file, unless an error occurs.

If an input/output item is a pointer, data transfers between the file and the associated target.

During advancing input from a file with a **PAD=** specifier that has the value **NO**, the input list and format specification must not require more characters from the record than that record contains. If the **PAD=** specifier has the value **YES**, blank characters are supplied if the input list and format specification require more characters from the record than the record contains.

<table>
<thead>
<tr>
<th>IBM Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>If you want to pad files connected for sequential access, specify the <strong>-qxf77=noblankpad</strong> compiler option. This compiler option also sets the default value for the <strong>PAD=</strong> specifier to <strong>NO</strong> for direct and stream files and <strong>YES</strong> for sequential files.</td>
</tr>
</tbody>
</table>

| End of IBM Extension |

During **nonadvancing input from** a file with a **PAD=** specifier that 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. If the record is the last record of a stream file, an **end-of-file condition** occurs.
Asynchronous Input/Output

IBM Extension

You can specify asynchronous **READ** and **WRITE** data transfer statements to initiate asynchronous data transfer. Execution continues after the asynchronous I/O statement, without waiting for the data transfer to complete. Executing a matching **WAIT** statement with the same **ID=** value that was returned to the **ID=** variable in the data transfer statement detects that the data transfer statement is complete, or waits for that data transfer statement to complete.

The data transfer of an I/O item in an asynchronous I/O statement can complete:
- During the execution of the asynchronous data transfer statement
- At any time before the execution of the matching **WAIT** statement
- During the matching **WAIT** statement

For information on situations where data transfer must complete during the asynchronous data transfer statement, see [Implementation details of XL Fortran Input/Output](#) in the [XL Fortran Optimization and Programming Guide](#).

If an error occurs during the execution of an asynchronous data transfer statement, the statement executes as if it were synchronous. The **ID=** specifier remains undefined and the accompanying **WAIT** statement does not execute. Instead of the **WAIT** statement, the **ERR=** specifier handles the error, and the **IOSTAT=** specifier indicates the status of the I/O operation.

You must not reference, define, or undefine variables or items associated with a variable appearing in an I/O list for an asynchronous data transfer statement, until the execution of the matching **WAIT** statement.

Any deallocation of allocatable objects and pointers and changing association status of pointers are disallowed between an asynchronous data transfer statement and the matching **WAIT** statement.

Multiple outstanding asynchronous data transfer operations on the same unit must all be **READ** or all be **WRITE**. You must not specify other I/O statements on the same unit until the matching **WAIT** statements for all outstanding asynchronous data transfer operations on the same unit execute. 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. **F2003** For stream access, an asynchronous **WRITE** statement must not specify either the same unit and location within a file as any asynchronous **WRITE** statement for which the matching **WAIT** statement has not been executed. **F2003**

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.

In the portion of the program that executes between the asynchronous data transfer statement and the matching **WAIT** statement, you must not reference, define, or undefine variables or items associated with the integer_variable in the **NUM=** specifier of a **READ** or **WRITE** statement.
Using Asynchronous I/O

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

Advancing and nonadvancing Input/Output
Advancing I/O positions the file after the last record that is read or written, unless
an error condition occurs.

Nonadvancing I/O can position the file at a character position within the current
record or a subsequent record. With nonadvancing I/O, you can READ or WRITE
a record of the file by a sequence of I/O statements that each access a portion of
the record. You can also read variable-length records and inquire about the length
of the records.

Nonadvancing I/O
! 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')))
         PRINT *, "Invalid character ", DIGIT, " at record position ", COUNT
STOP
END IF
END DO
100 PRINT *, "Number of digits in record = ", COUNT
END

! When the contents of fort.7 is '1234\n', the output is:
! Number of digits in record = 4

File position before and after data transfer
For an explicit connection using an `OPEN` statement for sequential or stream I/O that specifies the `POSITION=` specifier, you can position the file explicitly at the beginning, at the end, 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 position is at the beginning.

<table>
<thead>
<tr>
<th>IBM Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>• If you specify <code>STATUS='OLD'</code> with the <code>qposition=appendold</code> compiler option, and the next operation that changes the file position is a <code>WRITE</code> statement, then the file position is at the end. If these conditions are not met, the file position is at the beginning.</td>
</tr>
<tr>
<td>• If you specify <code>STATUS='UNKNOWN'</code> with the <code>qposition=appendunknown</code> compiler option, and the next operation is a <code>WRITE</code> statement, then the file position is at the end. If these conditions are not met, the file position is at the beginning.</td>
</tr>
</tbody>
</table>

After an implicit `OPEN` the file position is at the beginning:
• If the first I/O operation on the file is `PRINT` or `READ` the application reads the first record of the file.
• If the first I/O operation on the file is `WRITE` the application deletes the contents of the file and writes at the first record.

<table>
<thead>
<tr>
<th>End of IBM Extension</th>
</tr>
</thead>
</table>

You can use a `REWIND` statement to position a file at the beginning. The preconnected units 0, 5 and 6 are positioned as they come from the parent process of the application.

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 position is at the beginning of the next record. This record becomes the current record.
  – Advancing output creates a new record and becomes the last record of the file.
• Sequential access for an internal file:
  – File position is at the beginning of the first record of the file. This record becomes the current record.
• Direct access
  – File position is at the beginning of the record that the `REC=` specifier indicates. This record becomes the current record.
• Stream access
  • File position is immediately before the file storage unit the POS= specifier indicates. If there is no POS= specifier, the file position remains unchanged.

After advancing I/O data transfer the file position is:
• 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 or formatted stream access becomes the last record of the file.

After nonadvancing input the file position:
• If no error condition or end-of-file condition occurs, but an end-of-record condition occurs, the file position is immediately 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 does not change.
• In all other cases, the file position is immediately after the record read or written and that record becomes the preceding record.

If the file position is beyond the endfile record, a READ WRITE PRINT or ENDFILE statement can not execute if the compiler option -qxlf77=soteof is not set. A BACKSPACE or REWIND statement can be used to reposition the file.

---

IBM Extension

Use the -qxlf77=soteof option to be able to read and write past the end-of-file.

---

End of IBM Extension

Fortran 2003 Standard

For formatted stream output with no errors, the terminal point of the file is set to the highest-numbered position to which data was transferred by the statement. For unformatted stream output with no errors, the file position is unchanged. If the file position exceeds the previous terminal point of the file, the terminal point is set to the file position. Use the POS= specifier with an empty output list to extend the terminal point of the file without writing data. After data transfer, if an error occurs, the file position is indeterminate.

---

End of Fortran 2003 Standard

---

Conditions and IOSTAT values

An IOSTAT value is a value assigned to the variable for the IOSTAT= specifier if end-of-file condition or end-of-record condition or an error condition occurs during an input/output statement. The IOSTAT= specifier reports the following types of error conditions:
• Catastrophic
• Severe
End-of-record conditions

When an application encounters an end-of-record condition with the IOSTAT= specifier, it sets the value to -4 and branches to the EOR= label if that label is present. If the IOSTAT= and EOR= specifiers are not present on the I/O statement when an application encounters an end-of-record condition, the application stops.

Table 14. IOSTAT values for end-of-record conditions

<table>
<thead>
<tr>
<th>IOSTAT Value</th>
<th>End-of-Record Condition Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4</td>
<td>End of record encountered on a nonadvancing, format-directed READ of an internal or external file.</td>
</tr>
</tbody>
</table>

End-of-file conditions

An end-of-file condition can occur in the following instances:
- At the beginning of the execution of an input statement.
- During execution of a formatted input statement that requires more than one record through the interaction of the input list and the format.
- During execution of a stream input statement.
- When encountering an endfile record while reading of a file connected for sequential access.
- When attempting to read a record beyond the end of an internal file.

For stream access, an end-of-file condition occurs when you attempt to read beyond the end of a file. An end-of-file condition also occurs if you attempt to read beyond the last record of a stream file connected for formatted access.

An end-of-file condition causes IOSTAT= to be set to one of the values defined below and branches to the the END= label 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 15. IOSTAT values for end-of-file conditions

<table>
<thead>
<tr>
<th>IOSTAT Value</th>
<th>End-of-File Condition Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>End of file encountered on sequential or stream READ of an external file, or END= is specified on a direct access read and the record is nonexistent.</td>
</tr>
<tr>
<td>-1[SF7000000]</td>
<td>End of file encountered on READ of an internal file.</td>
</tr>
<tr>
<td>-2</td>
<td>End of file encountered on READ of an internal file.</td>
</tr>
</tbody>
</table>

Notes:
1. Fortran 2003 Standard. See the IOSTAT_END run-time option for more information.
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 can result, depending on how you configure your execution environment.

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 16. IOSTAT Values for severe error conditions

<table>
<thead>
<tr>
<th>IOSTAT Value</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>END= is not specified on a direct access READ and the record is nonexistent.</td>
</tr>
<tr>
<td>2</td>
<td>End of file encountered on WRITE of an internal file.</td>
</tr>
<tr>
<td>6</td>
<td>File cannot be found and STATUS='OLD' is specified on an OPEN statement.</td>
</tr>
<tr>
<td>10</td>
<td>Read error on direct file.</td>
</tr>
<tr>
<td>11</td>
<td>Write error on direct file.</td>
</tr>
<tr>
<td>12</td>
<td>Read error on sequential or stream file.</td>
</tr>
<tr>
<td>13</td>
<td>Write error on [sequential or stream] file.</td>
</tr>
<tr>
<td>14</td>
<td>Error opening file.</td>
</tr>
<tr>
<td>15</td>
<td>Permanent I/O error encountered on file.</td>
</tr>
<tr>
<td>37</td>
<td>Dynamic memory allocation failure - out of memory.</td>
</tr>
<tr>
<td>38</td>
<td>REWIND error.</td>
</tr>
<tr>
<td>39</td>
<td>ENDFILE error.</td>
</tr>
<tr>
<td>40</td>
<td>BACKSPACE error.</td>
</tr>
<tr>
<td>107</td>
<td>File exists and STATUS='NEW' was specified on an OPEN statement.</td>
</tr>
<tr>
<td>119</td>
<td>BACKSPACE statement attempted on unit connected to a tape device.</td>
</tr>
<tr>
<td>122</td>
<td>Incomplete record encountered during direct access READ.</td>
</tr>
<tr>
<td>130</td>
<td>ACTION='READWRITE' specified on an OPEN statement to connect a pipe.</td>
</tr>
<tr>
<td>135</td>
<td>The user program is making calls to an unsupported version of the XL Fortran run-time environment.</td>
</tr>
<tr>
<td>139</td>
<td>I/O operation not permitted on the unit because the file was not opened with an appropriate value for the ACTION= specifier.</td>
</tr>
<tr>
<td>142</td>
<td>CLOSE error.</td>
</tr>
<tr>
<td>144</td>
<td>INQUIRE error.</td>
</tr>
<tr>
<td>152</td>
<td>ACCESS='DIRECT' is specified on an OPEN statement for a file that can only be accessed sequentially.</td>
</tr>
</tbody>
</table>
Table 16. IOSTAT Values for severe error conditions (continued)

<table>
<thead>
<tr>
<th>IOSTAT Value</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>153</td>
<td>POSITION=’REWIND’ or POSITION=’APPEND’ is specified on an OPEN statement and the file is a pipe.</td>
</tr>
<tr>
<td>156</td>
<td>Invalid value for RECL= specifier on an OPEN statement.</td>
</tr>
<tr>
<td>159</td>
<td>External file input could not be flushed because the associated device is not seekable.</td>
</tr>
<tr>
<td>165</td>
<td>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.</td>
</tr>
<tr>
<td>169</td>
<td>The asynchronous I/O statement cannot be completed because the unit is connected for synchronous I/O only.</td>
</tr>
<tr>
<td>172</td>
<td>The connection failed because the file does not allow asynchronous I/O.</td>
</tr>
<tr>
<td>173</td>
<td>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.</td>
</tr>
<tr>
<td>174</td>
<td>The synchronous I/O statement cannot be completed because an earlier asynchronous I/O statement has not been completed.</td>
</tr>
<tr>
<td>175</td>
<td>The WAIT statement cannot be completed because the value of the ID= specifier is invalid.</td>
</tr>
<tr>
<td>176</td>
<td>The WAIT statement cannot be completed because the corresponding asynchronous I/O statement is in a different scoping unit.</td>
</tr>
<tr>
<td>178</td>
<td>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.</td>
</tr>
<tr>
<td>179</td>
<td>The I/O operation cannot be performed on the unit because there are still incomplete asynchronous I/O operations on the unit.</td>
</tr>
<tr>
<td>181</td>
<td>A file cannot be connected to a unit because multiple connections are allowed for synchronous I/O only.</td>
</tr>
<tr>
<td>182</td>
<td>Invalid value for UWIDTH= option. It must be set to either 32 or 64.</td>
</tr>
<tr>
<td>183</td>
<td>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.</td>
</tr>
<tr>
<td>184</td>
<td>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.</td>
</tr>
<tr>
<td>185</td>
<td>A file cannot be connected to two units with different UWIDTH values.</td>
</tr>
<tr>
<td>186</td>
<td>Unit numbers must be between 0 and 2,147,483,647.</td>
</tr>
<tr>
<td>192</td>
<td>The value of the file position is out of the range of the scalar variable specified with the POS= specifier in the INQUIRE statement.</td>
</tr>
<tr>
<td>193</td>
<td>The value of the file size is out of the range of the scalar variable specified with the SIZE= specifier in the INQUIRE statement.</td>
</tr>
<tr>
<td>200</td>
<td><strong>FLUSH</strong> error.</td>
</tr>
<tr>
<td>201</td>
<td>The unit specified in the <strong>FLUSH</strong> statement is connected to a non-seekable file.</td>
</tr>
</tbody>
</table>

**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 17. IOSTAT values for recoverable error conditions

<table>
<thead>
<tr>
<th>IOSTAT Value</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>Value of REC= specifier invalid on direct I/O.</td>
</tr>
<tr>
<td>17</td>
<td>I/O statement not allowed on direct file.</td>
</tr>
<tr>
<td>18</td>
<td>Direct I/O statement on an unconnected unit.</td>
</tr>
<tr>
<td>19</td>
<td>Unformatted I/O attempted on formatted file.</td>
</tr>
<tr>
<td>20</td>
<td>Formatted I/O attempted on unformatted file.</td>
</tr>
<tr>
<td>21</td>
<td>Sequential or stream I/O attempted on direct file.</td>
</tr>
<tr>
<td>22</td>
<td>Direct I/O attempted on sequential or stream file.</td>
</tr>
<tr>
<td>23</td>
<td>Attempt to connect a file that is already connected to another unit.</td>
</tr>
<tr>
<td>24</td>
<td>OPEN specifiers do not match the connected file’s attributes.</td>
</tr>
<tr>
<td>25</td>
<td>RECL= specifier omitted on an OPEN statement for a direct file.</td>
</tr>
<tr>
<td>26</td>
<td>RECL= specifier on an OPEN statement is negative.</td>
</tr>
<tr>
<td>27</td>
<td>ACCESS= specifier on an OPEN statement is invalid.</td>
</tr>
<tr>
<td>28</td>
<td>FORM= specifier on an OPEN statement is invalid.</td>
</tr>
<tr>
<td>29</td>
<td>STATUS= specifier on an OPEN statement is invalid.</td>
</tr>
<tr>
<td>30</td>
<td>BLANK= specifier on an OPEN statement is invalid.</td>
</tr>
<tr>
<td>31</td>
<td>FILE= specifier on an OPEN or INQUIRE statement is invalid.</td>
</tr>
<tr>
<td>32</td>
<td>STATUS=’SCRATCH’ and FILE= specifier specified on same OPEN statement.</td>
</tr>
<tr>
<td>33</td>
<td>STATUS=’KEEP’ specified on CLOSE statement when file was opened with STATUS=’SCRATCH’.</td>
</tr>
<tr>
<td>34</td>
<td>Value of STATUS= specifier on CLOSE statement is invalid.</td>
</tr>
<tr>
<td>36</td>
<td>Invalid unit number specified in an I/O statement.</td>
</tr>
<tr>
<td>47</td>
<td>A namelist input item was specified with one or more components of nonzero rank.</td>
</tr>
<tr>
<td>48</td>
<td>A namelist input item specified a zero-sized array.</td>
</tr>
<tr>
<td>58</td>
<td>Format specification error.</td>
</tr>
<tr>
<td>93</td>
<td>I/O statement not allowed on error unit (unit 0).</td>
</tr>
<tr>
<td>110</td>
<td>Illegal edit descriptor used with a data item in formatted I/O.</td>
</tr>
<tr>
<td>120</td>
<td>The NLWIDTH setting exceeds the length of a record.</td>
</tr>
<tr>
<td>125</td>
<td>BLANK= specifier given on an OPEN statement for an unformatted file.</td>
</tr>
<tr>
<td>127</td>
<td>POSITION= specifier given on an OPEN statement for a direct file.</td>
</tr>
<tr>
<td>128</td>
<td>POSITION= specifier value on an OPEN statement is invalid.</td>
</tr>
<tr>
<td>129</td>
<td>ACTION= specifier value on an OPEN statement is invalid.</td>
</tr>
<tr>
<td>131</td>
<td>DELIM= specifier given on an OPEN statement for an unformatted file.</td>
</tr>
<tr>
<td>132</td>
<td>DELIM= specifier value on an OPEN statement is invalid.</td>
</tr>
<tr>
<td>133</td>
<td>PAD= specifier given on an OPEN statement for an unformatted file.</td>
</tr>
</tbody>
</table>
Table 17. IOSTAT values for recoverable error conditions (continued)

<table>
<thead>
<tr>
<th>IOSTAT Value</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>134</td>
<td>PAD= specifier value on an OPEN statement is invalid.</td>
</tr>
<tr>
<td>136</td>
<td>ADVANCE= specifier value on a READ statement is invalid.</td>
</tr>
<tr>
<td>137</td>
<td>ADVANCE='NO' is not specified when SIZE= is specified on a READ statement.</td>
</tr>
<tr>
<td>138</td>
<td>ADVANCE='NO' is not specified when EOR= is specified on a READ statement.</td>
</tr>
<tr>
<td>145</td>
<td>READ or WRITE attempted when file is positioned after the endfile record.</td>
</tr>
<tr>
<td>163</td>
<td>Multiple connections to a file located on a non-random access device are not allowed.</td>
</tr>
<tr>
<td>164</td>
<td>Multiple connections with ACTION='WRITE' or ACTION='READWRITE' are not allowed.</td>
</tr>
<tr>
<td>170</td>
<td>ASYNCH= specifier value on an OPEN statement is invalid.</td>
</tr>
<tr>
<td>171</td>
<td>ASYNCH= specifier given on an OPEN statement is invalid because the FORM= specifier is set to FORMATTED.</td>
</tr>
<tr>
<td>177</td>
<td>The unit was closed while there were still incomplete asynchronous I/O operations.</td>
</tr>
<tr>
<td>191</td>
<td>The RECL= specifier is specified on an OPEN statement that has ACCESS='STREAM'.</td>
</tr>
<tr>
<td>194</td>
<td>The BACKSPACE statement specifies a unit connected for unformatted stream I/O.</td>
</tr>
<tr>
<td>195</td>
<td>POS= specifier on an I/O statement is less than one.</td>
</tr>
<tr>
<td>196</td>
<td>The stream I/O statement cannot be performed on the unit because the unit is not connected for stream access.</td>
</tr>
<tr>
<td>197</td>
<td>POS= specifier on an I/O statement for a unit connected to a non-seekable file.</td>
</tr>
<tr>
<td>198</td>
<td>Stream I/O statement on an unconnected unit.</td>
</tr>
</tbody>
</table>

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 18. IOSTAT values for conversion error conditions

<table>
<thead>
<tr>
<th>IOSTAT Value</th>
<th>Error Description</th>
<th>IOSTAT set if CNVERR=NO</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>End of record encountered on an unformatted file.</td>
<td>no</td>
</tr>
<tr>
<td>4</td>
<td>End of record encountered on a formatted external file using advancing I/O.</td>
<td>no</td>
</tr>
<tr>
<td>5</td>
<td>End of record encountered on an internal file.</td>
<td>no</td>
</tr>
</tbody>
</table>
Table 18. IOSTAT values for conversion error conditions (continued)

<table>
<thead>
<tr>
<th>IOSTAT Value</th>
<th>Error Description</th>
<th>IOSTAT set if CNVERR=NO</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Incorrect format of list-directed input found in an external file.</td>
<td>yes</td>
</tr>
<tr>
<td>8</td>
<td>Incorrect format of list-directed input found in an internal file.</td>
<td>yes</td>
</tr>
<tr>
<td>9</td>
<td>List-directed or NAMElIST data item too long for the internal file.</td>
<td>yes</td>
</tr>
<tr>
<td>41</td>
<td>Valid logical input not found in external file.</td>
<td>no</td>
</tr>
<tr>
<td>42</td>
<td>Valid logical input not found in internal file.</td>
<td>no</td>
</tr>
<tr>
<td>43</td>
<td>Complex value expected using list-directed or NAMElIST input in external file but not found.</td>
<td>no</td>
</tr>
<tr>
<td>44</td>
<td>Complex value expected using list-directed or NAMElIST input in internal file but not found.</td>
<td>no</td>
</tr>
<tr>
<td>45</td>
<td>NAMElIST item name specified with unknown or invalid derived-type component name in NAMElIST input.</td>
<td>no</td>
</tr>
<tr>
<td>46</td>
<td>NAMElIST item name specified with an invalid substring range in NAMElIST input.</td>
<td>no</td>
</tr>
<tr>
<td>49</td>
<td>List-directed or namelist input contained an invalid delimited character string.</td>
<td>no</td>
</tr>
<tr>
<td>56</td>
<td>Invalid digit found in input for B, O or Z format edit descriptors.</td>
<td>no</td>
</tr>
<tr>
<td>84</td>
<td>NAMElIST group header not found in external file.</td>
<td>yes</td>
</tr>
<tr>
<td>85</td>
<td>NAMElIST group header not found in internal file.</td>
<td>yes</td>
</tr>
<tr>
<td>86</td>
<td>Invalid NAMElIST input value found in external file.</td>
<td>no</td>
</tr>
<tr>
<td>87</td>
<td>Invalid NAMElIST input value found in internal file.</td>
<td>no</td>
</tr>
<tr>
<td>88</td>
<td>Invalid name found in NAMElIST input.</td>
<td>no</td>
</tr>
<tr>
<td>90</td>
<td>Invalid character in NAMElIST group or item name in input.</td>
<td>no</td>
</tr>
<tr>
<td>91</td>
<td>Invalid NAMElIST input syntax.</td>
<td>no</td>
</tr>
<tr>
<td>92</td>
<td>Invalid subscript list for NAMElIST item in input.</td>
<td>no</td>
</tr>
<tr>
<td>94</td>
<td>Invalid repeat specifier for list-directed or NAMElIST input in external file.</td>
<td>no</td>
</tr>
<tr>
<td>95</td>
<td>Invalid repeat specifier for list-directed or NAMElIST input in internal file.</td>
<td>no</td>
</tr>
<tr>
<td>96</td>
<td>Integer overflow in input.</td>
<td>no</td>
</tr>
<tr>
<td>97</td>
<td>Invalid decimal digit found in input.</td>
<td>no</td>
</tr>
<tr>
<td>98</td>
<td>Input too long for B, O or Z format edit descriptors.</td>
<td>no</td>
</tr>
<tr>
<td>121</td>
<td>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.</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Fortran 90, 95 and 2003 standard 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.

A Fortran 2003 Standard language error results from the use of XL Fortran extensions to the Fortran 2003 language standard that cannot be detected at compile time. A Fortran 2003 language error is considered a severe error when the LANGLVL run-time option has been specified with the value 2003STD and the ERR_RECOVERY run-time option has either not been set or is set to NO. If both LANGLVL=2003STD 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 19. IOSTAT Values for Fortran 90, 95, and 2003 Standard Language Error Conditions

<table>
<thead>
<tr>
<th>IOSTAT Value</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>53</td>
<td>Mismatched edit descriptor and item type in formatted I/O.</td>
</tr>
<tr>
<td>58</td>
<td>Format specification error.</td>
</tr>
<tr>
<td>140</td>
<td>Unit is not connected when the I/O statement is attempted. Only for READ, WRITE, PRINT, REWIND, and ENDFILE.</td>
</tr>
<tr>
<td>141</td>
<td>Two ENDFILE statements without an intervening REWIND or BACKSPACE on the unit.</td>
</tr>
<tr>
<td>151</td>
<td>The FILE= specifier is missing and the STATUS= specifier does not have a value of ‘SCRATCH’ on an OPEN statement.</td>
</tr>
<tr>
<td>187</td>
<td>NAMELIST comments are not allowed by the Fortran 90 standard.</td>
</tr>
<tr>
<td>199</td>
<td>STREAM is not a valid value for the ACCESS= specifier on an OPEN statement in Fortran 90 or Fortran 95.</td>
</tr>
</tbody>
</table>
Chapter 9. Input/Output formatting

Formatted READ, WRITE and PRINT data transfer statements use formatting information to direct the conversion between internal data representations and character representations in a formatted record. You can control the conversion process, called editing, by using a formatting type. The Formatting and Access Types table details the access types that support each formatting type.

Table 20. Formatting and access types

<table>
<thead>
<tr>
<th>Formatting Type</th>
<th>Access Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format-directed</td>
<td>sequential, direct, and stream</td>
</tr>
<tr>
<td>List-directed</td>
<td>sequential, and stream</td>
</tr>
<tr>
<td>Namelist</td>
<td>sequential, and stream</td>
</tr>
</tbody>
</table>

Editing occurs on all fields in a record. A field is the part of a record that is read on input or written on output when format control processes a data or character string edit descriptor. The field width is the size of that field in characters.

Format-directed formatting

Format-directed formatting allows you to control editing using edit descriptors in a format specification. Specify a format specification in a FORMAT statement or as the value of a character array or character expression in a data transfer statement. Edit descriptors allow you to control editing in the following ways:

- Data edit descriptors allow you to specify editing by data type.
- Control edit descriptors focus on the editing process.
- Character string edit descriptors control string outputs.

Complex editing

To edit complex values, you must specify complex editing by using a pair of data edit descriptors. A complex value is a pair of separate real components. When specifying complex editing, the first edit descriptor applies to the real part of the number. The second edit descriptor applies to the imaginary part of the number.

You can specify different edit descriptors for a complex editing pair and use one or more control edit descriptors between the edit descriptors in that pair. You must not specify data edit descriptors between the edit descriptors in that pair.

Data edit descriptors

Data edit descriptors allow you to specify editing by data type. You can use them to edit both character and numeric data. The Data Edit Descriptors table contains a complete list of all character, character string and numeric edit descriptors. Numeric data refers to integer, real, and complex values.

Table 21. Data edit descriptors

<table>
<thead>
<tr>
<th>Forms</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Edits character values</td>
</tr>
<tr>
<td>Aw</td>
<td></td>
</tr>
</tbody>
</table>
**Table 21. Data edit descriptors (continued)**

<table>
<thead>
<tr>
<th>Forms</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bu</td>
<td>Edits binary values</td>
</tr>
<tr>
<td>Bw.m</td>
<td></td>
</tr>
<tr>
<td>Ewad</td>
<td>Edits real and complex numbers with exponents</td>
</tr>
<tr>
<td>EwadEe</td>
<td></td>
</tr>
<tr>
<td>EwadDe</td>
<td></td>
</tr>
<tr>
<td>EwadQe</td>
<td></td>
</tr>
<tr>
<td>Dwad</td>
<td></td>
</tr>
<tr>
<td>ENwad</td>
<td></td>
</tr>
<tr>
<td>ENwadEe</td>
<td></td>
</tr>
<tr>
<td>ESwad</td>
<td></td>
</tr>
<tr>
<td>ESwadEe</td>
<td></td>
</tr>
<tr>
<td>Otw.d</td>
<td>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</td>
</tr>
<tr>
<td>Gtw.d</td>
<td></td>
</tr>
<tr>
<td>GtwdEe</td>
<td></td>
</tr>
<tr>
<td>GtwdDe</td>
<td></td>
</tr>
<tr>
<td>GtwdQe</td>
<td></td>
</tr>
<tr>
<td>Iw.m</td>
<td>Edits integer numbers</td>
</tr>
<tr>
<td>Iw</td>
<td></td>
</tr>
<tr>
<td>Lw.p</td>
<td>Edits logical values</td>
</tr>
<tr>
<td>Otw.m</td>
<td>Edits octal values</td>
</tr>
<tr>
<td>Qw.*</td>
<td>Returns the count of characters remaining in an input record *</td>
</tr>
<tr>
<td>Ztw.m</td>
<td>Edits hexadecimal values</td>
</tr>
</tbody>
</table>

where:

* specifies an IBM extension.

d  Specifies the number of digits to the right of the decimal point.

e  Specifies the number of digits in the exponent field.

m  Specifies the number of digits to print.

n  Specifies the number of characters in a literal field. Blanks are included in character count.

w  Specifies the width of a field including all blanks as a positive value.

If you specify the B, F, I, O, or Z edit descriptors on output, the value of w can be zero.

**Rules for Data Edit Descriptor and Modifiers**

You must not specify kind type parameters.

Edit descriptor modifiers must be unsigned integer literal constants.

---

**IBM Extension**

For the w, m, d, and e modifiers, you must enclose a scalar integer expression in angle brackets (< and >). See “Variable format expressions” on page 317 for details.

**Note:**
There are two types of Q data edit descriptor:

**extended precision Q**
- is the Q edit descriptor with the `Qw.d` syntax

**character count Q**
- is the Q edit descriptor with the `Q` syntax

---

### Rules for numeric edit descriptors on input

Leading blanks are not significant. You can control the interpretation of other blanks using the `BLANK=` specifier in the `OPEN` or `READ` statements and the `BN` and `BZ` edit descriptors. A field of all blanks is treated as zero.

Plus signs are optional, though you must not specify plus signs for the `B` 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 contain more digits than can be represented internally.

---

### Rules for numeric data edit descriptors on output

Characters are right-justified in the field.

When the number of characters in a field is less than the field width, leading blanks fill the remaining field space.

When the number of characters in a field is greater than the field width, or if an exponent exceeds its specified width, asterisks fill the entire field space.

A minus sign prefixes a negative value. A positive or zero value does not receive a plus sign prefix on output, unless you specify the `S`, `SP`, or `SS` edit descriptors.

---

### Fortran 95

If you specify the `-qxlf90` compiler option the `E`, `D`, `Q(Extended Precision)`, `F`, `EN`, `ES`, and `G(General Editing)` edit descriptors output a negative value differently depending on the `signedzero` suboption.

- If you specify the `signedzero` suboption, the output field contains a minus sign for a negative value, even if that value is negative zero. This behavior conforms to the Fortran 95 and Fortran 2003 Standard.

---

### IBM Extension

XL Fortran does not evaluate a `REAL(16)` internal value of zero as a negative zero.

- If you specify the `nosignedzero` suboption, a minus sign is not written to the output field for a value of zero, even if the internal value is negative.
The \texttt{EN} and \texttt{ES} edit descriptors output a minus sign when the value is negative for the \texttt{signedzero} and \texttt{nospreadzero} suboptions.

---

**End of Fortran 95**

---

**IBM Extension**

XL Fortran indicates, a \texttt{NaN} (not a number) by “\texttt{NAN}”, “\texttt{+NAN}”, or “\texttt{-NAN}”. XL Fortran indicates infinity by “\texttt{INF}”, “\texttt{+INF}”, or “\texttt{-INF}”.

---

**Control edit descriptors**

*Table 22. Control edit descriptors*

<table>
<thead>
<tr>
<th>Forms</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>/ \</td>
<td>Specifies the end of data transfer on the current record</td>
</tr>
<tr>
<td>: \</td>
<td>Specifies the end of format control if there are no more items in the input/output list</td>
</tr>
<tr>
<td>\S* \</td>
<td>Suppresses end-of-record in output *</td>
</tr>
<tr>
<td>\BN \</td>
<td>Ignores nonleading blanks in numeric input fields</td>
</tr>
<tr>
<td>\BZ \</td>
<td>Interprets nonleading blanks in numeric input fields as zeros</td>
</tr>
<tr>
<td>\kP \</td>
<td>Specifies a scale factor for real and complex items</td>
</tr>
<tr>
<td>\SS \</td>
<td>Specifies that plus signs are not to be written</td>
</tr>
<tr>
<td>\SP \</td>
<td>Specifies that plus signs are to be written</td>
</tr>
<tr>
<td>\Tc \</td>
<td>Specifies the absolute position in a record from which, or to which, the next character is transferred</td>
</tr>
<tr>
<td>\TLc \</td>
<td>Specifies the relative position (backward from the current position in a record) from which, or to which, the next character is transferred</td>
</tr>
<tr>
<td>\oX \</td>
<td>Specifies the relative position (forward from the current position in a record) from which, or to which, the next character is transferred</td>
</tr>
</tbody>
</table>

where:

- \* specifies an IBM extension.
- \( r \) 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.

**Rules for Control Edit Descriptors and Modifiers**

You must not specify kind type parameters.
Character string edit descriptors

Character string edit descriptors allow you to edit character data.

<table>
<thead>
<tr>
<th>Forms</th>
<th>Use</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>nHstr</td>
<td>Outputs a character string (str)</td>
<td>211</td>
</tr>
<tr>
<td>'str'</td>
<td>Outputs a character string (str)</td>
<td>199</td>
</tr>
<tr>
<td>&quot;str&quot;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( n \) 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.

**Apostrophe/Double quotation mark editing**

**Purpose:** The apostrophe/double quotation mark edit descriptor specifies a character literal constant in an output format specification.

**Syntax:**
- `'character string'`
- `"character string"`

**Rules:** The width of the output field is the length of the character literal constant. See "Character" on page 25 for additional information on character literal constants.

**Notes:**
1. A backslash is recognized, by default, as an escape sequence, and as a backslash character when the -qnoescape compiler option is specified. See escape sequences for more information.
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:
```
ITIME=8
WRITE(*,5) ITIME
5 FORMAT('The value is -- ',I2) ! The value is -- 8
```
 Interaction of Input/Output lists and format specifications

Beginning format-directed formatting initiates format control. Each action of format control depends on the next edit descriptor 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 or an implied-DO list with an iteration count of zero. 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. An list item of complex type requires the interpretation of two F, E, or 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.

Comma-separated Input/Output

When reading floating-point data using format-directed input/output, a comma that appears in the input terminates the field. This can be useful for reading files containing comma-separated values.

For example, the following program reads two reals using the E edit descriptor. It requires that the field width be 16 characters. The program attempts to read the remaining characters in the record as a character string.

```fortran
> cat read.f
real a,b
character*10 c
open(11, access='sequential', form='formatted')
read(11, '(2e16.10, A)') a,b,c
print *, a
print *, b
print *, c
end
```

If the floating-point fields are 16 characters wide, as the format specifies, the program executes correctly. (0.400000000E+02 is 16 characters long.)

```fortran
> cat fort.11
0.4000000000E+02 0.3000000000E+02 hello
> a.out
40.00000000
30.00000000
hello
```

But if the floating-point input contains less than 16 characters, errors occur because parts of the next field are read. (0.400000E+02 is 12 characters long.)

```fortran
> cat fort.11
0.400000E+02 0.3000000E+02 hello
> a.out
1525-097 A READ statement using decimal base input found the invalid digit '.' in the input file.
The program will recover by assuming a zero in its place.
1525-097 A READ statement using decimal base input found the invalid digit 'h' in the input file.
The program will recover by assuming a zero in its place.
```
A READ statement using decimal base input found the invalid digit 'e' in the input file. The program will recover by assuming a zero in its place.

A READ statement using decimal base input found the invalid digit 'l' in the input file. The program will recover by assuming a zero in its place.

A READ statement using decimal base input found the invalid digit 'o' in the input file. The program will recover by assuming a zero in its place.

INF
0.0000000000E+00

If you use commas to terminate the fields, the floating-point values are read correctly. (0.400000E+02 is 12 characters long, but the fields are separated by commas.)

```
> cat fort.11
0.400000E+02,0.3000000E+02,hello
> a.out
40.00000000
30.00000000
hello
```

--- End of IBM Extension ---

**Data edit descriptors**

In the examples of data edit descriptors, a lowercase b in the Output column indicates that a blank appears at that position.

**A (Character) Editing**

**Purpose**
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.

**Syntax**

- `A`
- ` Aw`

**Rules**

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.
During formatted stream access, character output is split across more than one record if it contains newline characters.

**B (Binary) Editing**

**Purpose**
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.)

**Syntax**
- \( B^w \)
- \( B^{w.m} \)

**Rules**
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.

**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.

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 `noolboz` 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 `BLANK=` specifier and the `BN` and `BZ` edit descriptors affect the B edit descriptor.

**IBM Extension**
If the `oldboz` suboption of the `-qxlf77` compiler option is specified, the following occurs on output:
• $Bw$ is treated as $Bwm$, 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 BLANK= specifier and the BN and BZ edit descriptors do not affect the B edit descriptor.

---

**Examples**

**Examples of B editing on input:**

<table>
<thead>
<tr>
<th>Input</th>
<th>Format</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>B3</td>
<td>7</td>
</tr>
<tr>
<td>110</td>
<td>B3</td>
<td>6</td>
</tr>
</tbody>
</table>

**Examples of B editing on output:**

<table>
<thead>
<tr>
<th>Value</th>
<th>Format</th>
<th>Output (with -qxlf77=oldboz)</th>
<th>Output (with -qxlf77=nooldboz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>B3</td>
<td>111</td>
<td>111</td>
</tr>
<tr>
<td>6</td>
<td>B5</td>
<td>00110</td>
<td>bb110</td>
</tr>
<tr>
<td>17</td>
<td>B6.5</td>
<td>b10001</td>
<td>b10001</td>
</tr>
<tr>
<td>17</td>
<td>B4.2</td>
<td>0001</td>
<td>****</td>
</tr>
<tr>
<td>22</td>
<td>B6.5</td>
<td>b10110</td>
<td>b10110</td>
</tr>
<tr>
<td>22</td>
<td>B4.2</td>
<td>0110</td>
<td>****</td>
</tr>
<tr>
<td>0</td>
<td>B5.0</td>
<td>bbbbb</td>
<td>bbbbb</td>
</tr>
<tr>
<td>2</td>
<td>B0</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

---

**E, D, and Q (Extended Precision) Editing**

**Purpose**

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 in XL Fortran, as long as the length is at least 4 bytes.

**Syntax**

- $Ew.d$
- $Ew.d Ee$
- $Dw.d$

**Rules**

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—decimal_exponent
```

`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):

<table>
<thead>
<tr>
<th>Edit Descriptor</th>
<th>Absolute Value of Exponent (with scale factor of 0)</th>
<th>Form of Exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>E(__d)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E(__d)</td>
<td>(</td>
<td>\text{decimal_exponent}</td>
</tr>
<tr>
<td>E(__d)</td>
<td>(99 &lt;</td>
<td>\text{decimal_exponent}</td>
</tr>
<tr>
<td>E(__d)De</td>
<td>(</td>
<td>\text{decimal_exponent}</td>
</tr>
<tr>
<td>E(__d)De*</td>
<td>(</td>
<td>\text{decimal_exponent}</td>
</tr>
<tr>
<td>E(__d)Qe</td>
<td>(</td>
<td>\text{decimal_exponent}</td>
</tr>
<tr>
<td>E(__d)Qe</td>
<td>(99 &lt;</td>
<td>\text{decimal_exponent}</td>
</tr>
<tr>
<td>Q(__d)</td>
<td>(</td>
<td>\text{decimal_exponent}</td>
</tr>
<tr>
<td>Q(__d)</td>
<td>(99 &lt;</td>
<td>\text{decimal_exponent}</td>
</tr>
</tbody>
</table>

**Note:** * IBM Extensions

The scale factor \( k \) (see “P (Scale Factor) Editing” on page 221) controls decimal normalization. If \(-d< k \leq 0\), 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 see “Rules for numeric edit descriptors on input” on page 197.

**IBM Extension**

**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>
<thead>
<tr>
<th>Display</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAN</td>
<td>Positive Quiet NaN (not-a-number)</td>
</tr>
<tr>
<td>+NAN</td>
<td>Positive Quiet NaN (not-a-number)</td>
</tr>
<tr>
<td>-NAN</td>
<td>Negative Quiet NaN</td>
</tr>
<tr>
<td>NAN</td>
<td>Positive Signaling NaN</td>
</tr>
<tr>
<td>+NAN</td>
<td>Positive Signaling NaN</td>
</tr>
</tbody>
</table>

Table 23. Floating-point display
Table 23. Floating-point display (continued)

<table>
<thead>
<tr>
<th>Display</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-NAN</td>
<td>Negative Signaling NaN</td>
</tr>
<tr>
<td>INF</td>
<td>Positive Infinity</td>
</tr>
<tr>
<td>+INF</td>
<td>Negative Infinity</td>
</tr>
</tbody>
</table>

---

**Examples**

**Examples of E, D, and extended precision Q editing on input:** (Assume BN editing is in effect for blank interpretation.)

<table>
<thead>
<tr>
<th>Input</th>
<th>Format</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.34</td>
<td>E8.4</td>
<td>12.34</td>
</tr>
<tr>
<td>.1234E2</td>
<td>E8.4</td>
<td>12.34</td>
</tr>
<tr>
<td>2.E10</td>
<td>E12.6E1</td>
<td>2.E10</td>
</tr>
</tbody>
</table>

**Examples of E, D, and extended precision Q editing on output:**

<table>
<thead>
<tr>
<th>Value</th>
<th>Format</th>
<th>Output (with -qxlf77=noleadzero)</th>
<th>Output (with -qxlf77=leadzero)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234.56</td>
<td>E10.3</td>
<td>bb.123E+04</td>
<td>b0.123E+04</td>
</tr>
<tr>
<td>1234.56</td>
<td>D10.3</td>
<td>bb.123D+04</td>
<td>b0.123D+04</td>
</tr>
</tbody>
</table>

**Fortran 95**

<table>
<thead>
<tr>
<th>Value</th>
<th>Format</th>
<th>Output (with -qxlf90=signedzero)</th>
<th>Output (with -qxlf90=nosignedzero)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.001</td>
<td>E5.2</td>
<td>-0.00</td>
<td>b0.00</td>
</tr>
</tbody>
</table>

---

**End of IBM Extension**

**EN Editing**

**Purpose**

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 in XL Fortran, as long as the length is at least 4 bytes.

**Syntax**

- ENw.d
- ENw.dEe

**Rules**

The form and interpretation of the input field is the same as for F editing.

The form of the output field is:
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 \( 1 \leq yyy < 1000 \) 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):

<table>
<thead>
<tr>
<th>Edit Descriptor</th>
<th>Absolute Value of Exponent</th>
<th>Form of Exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENw.d</td>
<td>(</td>
<td>exp</td>
</tr>
<tr>
<td>ENw.d</td>
<td>( 99 &lt;</td>
<td>exp</td>
</tr>
<tr>
<td>ENw.dEe</td>
<td>(</td>
<td>exp</td>
</tr>
</tbody>
</table>

For additional information on numeric editing, see the “Rules for numeric edit descriptors on input” on page 197.

Examples

<table>
<thead>
<tr>
<th>Value</th>
<th>Format</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.14159</td>
<td>EN12.5</td>
<td>3.14159E+00</td>
</tr>
<tr>
<td>1.414250+5</td>
<td>EN15.5E4</td>
<td>141.42500E+003</td>
</tr>
<tr>
<td>3.141590-12</td>
<td>EN15.5E1</td>
<td>***************</td>
</tr>
</tbody>
</table>

Fortran 95

(with \(-qxlf90=signedzero\)) (with \(-qxlf90=nosignedzero\))

\(-0.001\) EN9.2 \(-1.00E-03\) \(-1.00E-03\)

End of Fortran 95

ES Editing

Purpose

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 in XL Fortran, as long as the length is at least 4 bytes.

Syntax

- \( ES\_w.d \)
- \( ES\_w.dEe \)
**Rules**
The form and interpretation of the input field is the same as for F editing.

The form of the output field is:

```
[ + ] - y ..digit_string—exp
```

- `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):

<table>
<thead>
<tr>
<th>Edit Descriptor</th>
<th>Absolute Value of Exponent</th>
<th>Form of Exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESw.d</td>
<td>$</td>
<td>\exp</td>
</tr>
<tr>
<td>ESw.d</td>
<td>$99 &lt;</td>
<td>\exp</td>
</tr>
<tr>
<td>ESw.dEe</td>
<td>$</td>
<td>\exp</td>
</tr>
</tbody>
</table>

For additional information on numeric editing, see "Rules for numeric edit descriptors on input" on page 197.

**Examples**

<table>
<thead>
<tr>
<th>Value</th>
<th>Format</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>31415.9</td>
<td>ES12.5</td>
<td>b3.14159E+04</td>
</tr>
<tr>
<td>14142.50+3</td>
<td>ES15.5E4</td>
<td>bb1.41425E+0007</td>
</tr>
<tr>
<td>31415.90-22</td>
<td>ES15.5E1</td>
<td>***************</td>
</tr>
</tbody>
</table>

***Fortran 95***

(with -qxlf90=signedzero) (with -qxlf90=nosignedzero)

-0.001 ES9.2 -1.00E-03 -1.00E-03

---

End of Fortran 95

**F (Real without Exponent) Editing**

**Purpose**
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 in XL Fortran, as long as the length is at least 4 bytes.

**Syntax**
- $Fw.d$
Rules
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 221 for more
   information.

See “Rules for numeric edit descriptors on input” on page 197 for additional
information.

On input, w must be greater than zero.

Fortran 95

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.

End of Fortran 95

Examples

Examples of F editing on input: (Assume BN editing is in effect for blank
interpretation.)

<table>
<thead>
<tr>
<th>Input</th>
<th>Format</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>-100</td>
<td>F6.2</td>
<td>-1.0</td>
</tr>
<tr>
<td>2.9</td>
<td>F6.2</td>
<td>2.9</td>
</tr>
<tr>
<td>4.E+2</td>
<td>F6.2</td>
<td>400.0</td>
</tr>
</tbody>
</table>

Examples of F editing on output:

<table>
<thead>
<tr>
<th>Value</th>
<th>Format</th>
<th>Output</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>(with -qxlf77=noleadzero)</td>
<td>(with -qxlf77=leadzero)</td>
</tr>
<tr>
<td>+1.2</td>
<td>F8.4</td>
<td>bb1.2000</td>
<td>bb1.2000</td>
</tr>
<tr>
<td>.12345</td>
<td>F8.3</td>
<td>bbb.123</td>
<td>bbb0.123</td>
</tr>
<tr>
<td>-12.34</td>
<td>F6.2</td>
<td>-12.34</td>
<td>-12.34</td>
</tr>
</tbody>
</table>

Fortran 95

-12.34  F0.2   -12.34   -12.34

-0.001  F5.2   -0.00    b0.00

Chapter 9. Input/Output formatting  209


## G (General) Editing

### Purpose
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.

### Syntax
- \( Gw.d \)
- \( Gw.dEe \)
- \( Gw.dQe \)

### Rules

**Generalized real and complex editing:** If the nogedit77 suboption (the default) of the \(-qxf77\) 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 \( 0 < N < 0.1-0.5 \times 10^{-d-1} \) or \( N \geq 10^{-d} - 0.5 \) or \( N \) is 0 and \( d \) is 0, \( Gw.d \) output editing is the same as \( kPE \ w.d \) output editing and \( Gw.dEe \) output editing is the same as \( kPEw.dEe \) output editing, where \( kP \) refers to the scale factor ("P (Scale Factor) Editing" on page 221) currently in effect. If \( 0.1-0.5 \times 10^{-d-1} \leq N < 10^d-0.5 \) 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:

<table>
<thead>
<tr>
<th>Magnitude of Datum</th>
<th>Equivalent Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N = 0 )</td>
<td>F(( w-n ),( d-1 ),( n )('b')) ( (d ) must not be 0)</td>
</tr>
<tr>
<td>( 0.1-0.5 \times 10^{-d-1} \leq N &lt; 1-0.5 \times 10^{-d} )</td>
<td>F(( w-n ),( d ),( n )('b'))</td>
</tr>
<tr>
<td>( 1-0.5 \times 10^{-d} \leq N &lt; 10-0.5 \times 10^{-d+1} )</td>
<td>F(( w-n ),( d-1 ),( n )('b'))</td>
</tr>
<tr>
<td>( 10-0.5 \times 10^{-d+1} \leq N &lt; 100-0.5 \times 10^{-d+2} )</td>
<td>F(( w-n ),( d-2 ),( n )('b'))</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( 10^{d-2}-0.5 \times 10^{-2} \leq N &lt; 10^{d-1} - 0.5 \times 10^{-1} )</td>
<td>F(( w-n ),( d-1 ),( n )('b'))</td>
</tr>
<tr>
<td>( 10^{d-1}-0.5 \times 10^{-1} \leq N &lt; 10^d-0.5 )</td>
<td>F(( w-n ),( d ),( n )('b'))</td>
</tr>
</tbody>
</table>

where \( b \) is a blank. \( n \) is 4 for \( Gw.d \) and \( e+2 \) for \( Gw.dEe \). 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.

### IBM Extension

If \( 0 < N < 0.1-0.5 \times 10^{-d-1} \), \( N \geq 10^d-0.5 \), or \( N \) is 0 and \( d \) is 0, \( Gw.dEe \) output editing is the same as \( kPEw.dEe \) output editing and \( Gw.dQe \) output editing is the
same as kPEw.dQe output editing.

End of IBM Extension

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 \( N \geq 10^d \):
  - \( Gw.d \) editing is the same as \( Ew.d \) editing
  - \( Gw.dE \) editing is the same as \( Ew.dE \) editing.

- If \( N \geq 0.1 \) and \( N < 10^d \):

<table>
<thead>
<tr>
<th>Magnitude of Datum</th>
<th>Equivalent Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0.1 \leq N &lt; 1 )</td>
<td>( F(w-n).d, n(b') )</td>
</tr>
<tr>
<td>( 1 \leq N &lt; 10 )</td>
<td>( F(w-n)(d-1), n(b') )</td>
</tr>
<tr>
<td>( 10^{d-2} \leq N &lt; 10^{d-1} )</td>
<td>.</td>
</tr>
<tr>
<td>( 10^{d-1} \leq N &lt; 10^d )</td>
<td>( F(w-n).1, n(b') )</td>
</tr>
<tr>
<td></td>
<td>( F(w-n).0, n(b') )</td>
</tr>
</tbody>
</table>

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 “Rules for numeric edit descriptors on input” on page 197 for additional information.

Examples

<table>
<thead>
<tr>
<th>Value</th>
<th>Format</th>
<th>Output (with -qxlf77=gedit77)</th>
<th>Output (with -qxlf77=nogedit77)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>G10.2</td>
<td>bb0.00E+00</td>
<td>bb0.0</td>
</tr>
<tr>
<td>0.0995</td>
<td>G10.2</td>
<td>bb0.10E+00</td>
<td>bb0.10</td>
</tr>
<tr>
<td>99.5</td>
<td>G10.2</td>
<td>bb100.</td>
<td>bb0.10E+03</td>
</tr>
</tbody>
</table>

H Editing

Purpose

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.

Syntax

- \( nH \ str \)

Rules

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:

--- IBM Extension ---

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 24 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.

--- End of IBM Extension ---

--- Fortran 95 ---

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 746 for more information.

--- End of Fortran 95 ---

Examples

```
50 FORMAT(16HThe value is -- ,I2)
10 FORMAT(I2,7Ho'clock)
   WRITE(*,'(I2,7Ho'clock)') ITIME
```

I (Integer) Editing

Purpose
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 <IBM> or any other type in XL Fortran. <IBM>

Syntax
- `IW`
- `IW.m`

Rules
- `w` includes the optional sign.
- `m` must have a value that is less than or equal to `w`, <F95> unless `w` is zero in Fortran 95. <F95>

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 [editing](#).

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**

**Examples of I editing on input:** (Assume BN editing is in effect for blank interpretation.)

<table>
<thead>
<tr>
<th>Input</th>
<th>Format</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>-123</td>
<td>I6</td>
<td>-123</td>
</tr>
<tr>
<td>123456</td>
<td>I7.5</td>
<td>123456</td>
</tr>
<tr>
<td>1234</td>
<td>I4</td>
<td>1234</td>
</tr>
</tbody>
</table>

**Examples of I editing on output:**

<table>
<thead>
<tr>
<th>Value</th>
<th>Format</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>-12</td>
<td>I7.6</td>
<td>-000012</td>
</tr>
<tr>
<td>12345</td>
<td>I5</td>
<td>12345</td>
</tr>
</tbody>
</table>

**L (Logical) Editing**

**Purpose**
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 in XL Fortran.

**Syntax**
- \( Lw \)
Rules
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
Examples of L editing on input:

<table>
<thead>
<tr>
<th>Input</th>
<th>Format</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>L4</td>
<td>true</td>
</tr>
<tr>
<td>.FALSE.</td>
<td>L7</td>
<td>false</td>
</tr>
</tbody>
</table>

Examples of L editing on output:

<table>
<thead>
<tr>
<th>Value</th>
<th>Format</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>L4</td>
<td>bbbT</td>
</tr>
<tr>
<td>FALSE</td>
<td>L1</td>
<td>F</td>
</tr>
</tbody>
</table>

O (Octal) Editing

Purpose
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.)

Syntax
- Ow
- Ow.m

Rules
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.

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.

End of Fortran 95

The output field for Ow 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 Ow.m is the same as for Ow, 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 BLANK= specifier and the BN and BZ edit descriptors affect the O edit descriptor.

IBM Extension

If the oldboz suboption of the -qxlf77 compiler option is specified, the following occurs on output:

- Ow is treated as Ow.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 BLANK= specifier and the BN and BZ edit descriptors do not affect the O edit descriptor.

End of IBM Extension

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

Examples of O editing on input:

<table>
<thead>
<tr>
<th>Input</th>
<th>Format</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>03</td>
<td>83</td>
</tr>
<tr>
<td>120</td>
<td>03</td>
<td>80</td>
</tr>
</tbody>
</table>

Examples of O editing on output:

<table>
<thead>
<tr>
<th>Value</th>
<th>Format</th>
<th>Output (with -qxlf77=oldboz)</th>
<th>Output (with -qxlf77=nooldboz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>05</td>
<td>00120</td>
<td>bb120</td>
</tr>
<tr>
<td>83</td>
<td>02</td>
<td>23</td>
<td>**</td>
</tr>
</tbody>
</table>

Fortran 95

<table>
<thead>
<tr>
<th>Value</th>
<th>Format</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>05.0</td>
<td>bbbbb</td>
</tr>
<tr>
<td>0</td>
<td>00.0</td>
<td>b</td>
</tr>
<tr>
<td>80</td>
<td>00</td>
<td>120</td>
</tr>
</tbody>
</table>

End of Fortran 95

Q (Character Count) Editing

IBM Extension

Purpose

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.
Syntax

- Q

Rules

There also exists the extended precision Q edit descriptor. By default, XL Fortran only recognizes the extended precision Q edit descriptor described earlier. See “E, D, and Q (Extended Precision) Editing” on page 204 for more information. To enable both Q edit descriptors, you must specify the \texttt{-qqcount} compiler option.

When you specify the \texttt{-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 \texttt{Qwo.} or \texttt{Qwo.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 \texttt{RECL=} specifier in the OPEN statement.
- Formatted stream external files. A record of this file type is terminated by a new-line character. Records in the same file have different lengths.

In an output operation, the character count Q edit descriptor is ignored. The corresponding output item is skipped.

Examples

\texttt{@PROCESS QCOUNT}

\texttt{CHARACTER(50) BUF}
\texttt{INTEGER(4) NBYTES}
\texttt{CHARACTER(60) STRING}

\texttt{...}
\texttt{BUF = 'This string is 29 bytes long.'}
\texttt{READ( BUF, FMT='(Q)' ) NBYTES}
\texttt{WRITE( *,* ) NBYTES}
\texttt{! NBYTES equals 50 because the buffer BUF is 50 bytes long.}
\texttt{READ(*,20) NBYTES, STRING}
\texttt{20 FORMAT(Q,A)}
\texttt{! NBYTES will equal the number of characters entered by the user.}
\texttt{END}

--- End of IBM Extension ---
**Z (Hexadecimal) Editing**

**Purpose**
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.)

**Syntax**
- **Zw**
- **Zw.m**

**Rules**
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.

<table>
<thead>
<tr>
<th>Fortran 95</th>
</tr>
</thead>
</table>

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.

| End of Fortran 95 |

The output field for **Zw** 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 **Zw.m** is the same as for **Zw**, 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.

| Fortran 95 |

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.

| End of Fortran 95 |

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 **BLANK** specifier and the **BN** and **BZ** edit descriptors affect the **Z** edit descriptor.

| IBM Extension |

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 BLANK= specifier and the BN and BZ edit descriptors do not affect the Z edit descriptor.

\[
\text{End of IBM Extension}
\]

**Examples**

**Examples of Z editing on input:**

<table>
<thead>
<tr>
<th>Input</th>
<th>Format</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0C</td>
<td>Z2</td>
<td>12</td>
</tr>
<tr>
<td>7FFF</td>
<td>Z4</td>
<td>32767</td>
</tr>
</tbody>
</table>

**Examples of Z editing on output:**

<table>
<thead>
<tr>
<th>Value</th>
<th>Format</th>
<th>Output</th>
<th>Output (with -qx1f77=oldboz)</th>
<th>Output (with -qx1f77=nooldboz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Z2</td>
<td>FF</td>
<td>**</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Z4</td>
<td>000C</td>
<td>bbbC</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Z0</td>
<td>C</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Z5.0</td>
<td>bbbbb</td>
<td>bbbbb</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Z0.0</td>
<td>b</td>
<td>b</td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{End of Fortran 95}
\]

**Control edit descriptors**

/ (Slash) Editing

**Purpose**

The slash edit descriptor indicates the end of data transfer on the current record. The repeat specifier ($r$) has a default value of 1.

**Syntax**

- `/`
- `/r/`

**Rules**

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.

---

**Fortran 2003 Standard**

When you connect a file for input using stream access, each slash edit descriptor positions the file at the beginning of the next record, skipping the remaining portion of the current record. On output to a file connected for stream access, a newly created empty record follows the current record. The new record becomes both the current and last record of the file, with the file position coming at the beginning of the new record.

---

**Examples**

```fortran
500  FORMAT(F6.2 / 2F6.2)
100  FORMAT(3/)
```

---

### : (Colon) Editing

**Purpose**
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.

**Syntax**
- `:`

**Rules**
See “Interaction of Input/Output lists and format specifications” on page 200 for more information.

**Examples**

```fortran
10   FORMAT(3(:'Array Value',F10.5)/)
```

---

### $ (Dollar) Editing

---

**IBM Extension**

**Purpose**
The dollar edit descriptor inhibits an end-of-record for a sequential or formatted stream WRITE statement.

**Syntax**
- `$`

**Rules**
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.
Examples
A common use for dollar sign editing is to prompt for a response and read the answer from the same line.

```
WRITE(*,FMT='($,A)') 'Enter your age '
READ(*,FMT='(BN,13)') IAGE
WRITE(*,FMT=1000)
1000 FORMAT('Enter your height: ',$)
READ(*,FMT='(F6.2)') HEIGHT
```

BN (Blank Null) and BZ (Blank Zero) Editing

Purpose
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.

Syntax
- BN
- BZ

Rules
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.

IBM Extension

If you specify the oldboz suboption of the –qxf77 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.
P (Scale Factor) Editing

Purpose
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.

Syntax
- \( kP \)

Rules
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
Examples of P editing on input:

<table>
<thead>
<tr>
<th>Input</th>
<th>Format</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>98.765</td>
<td>3P,F8.6</td>
<td>.98765E-1</td>
</tr>
<tr>
<td>98.765</td>
<td>-3P,F8.6</td>
<td>98765.</td>
</tr>
<tr>
<td>.98765E+2</td>
<td>3P,F10.5</td>
<td>.98765E+2</td>
</tr>
</tbody>
</table>

Examples of P editing on output:

<table>
<thead>
<tr>
<th>Value</th>
<th>Format</th>
<th>Output (with -qxlf77=noleadzero)</th>
<th>Output (with -qxlf77=leadzero)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.67</td>
<td>-3P,F7.2</td>
<td>bbbb.01</td>
<td>bbb0.01</td>
</tr>
<tr>
<td>12.34</td>
<td>-2P,F6.4</td>
<td>b.1234</td>
<td>0.1234</td>
</tr>
<tr>
<td>12.34</td>
<td>2P,E10.3</td>
<td>b12.34E+00</td>
<td>b12.34E+00</td>
</tr>
</tbody>
</table>

S, SP, and SS (Sign Control) Editing

Purpose
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.

Syntax
- S
- SP
- SS
Rules
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
<table>
<thead>
<tr>
<th>Value</th>
<th>Format</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.3456</td>
<td>S,F8.4</td>
<td>b12.3456</td>
</tr>
<tr>
<td>12.3456</td>
<td>SS,F8.4</td>
<td>b12.3456</td>
</tr>
<tr>
<td>12.3456</td>
<td>SP,F8.4</td>
<td>+12.3456</td>
</tr>
</tbody>
</table>

T, TL, TR, and X (Positional) Editing

Purpose
The T, TL, TR, and X edit descriptors specify the position where the transfer of the next character to or from a record starts.

Syntax
- \( Tc \)
- \( TLc \)
- \( TRc \)
- \( oX \)

Rules
The T and TL edit descriptors use the left tab limit for file positioning. Immediately before data transfer the definition of the left tab limit is the character position of the current record or the current position of the stream file. The T, TL, TR, and X specify the character position as follows:
- For \( Tc \), the \( c \)th character position of the record, relative to the left tab limit.
- For \( TLc \), \( c \) characters backward from the current position unless \( c \) is greater than the difference between the current character position and the left tab limit. Then, transmission of the next character to or from the record occurs at the left tab limit.
- For \( TRc \), \( c \) characters forward from the current position.
- For \( oX \), \( o \) 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.

IBM Extension

The X edit descriptor can be specified without a character position. It is treated as \( 1X \). When the source file is compiled with \(-qlanglvl=90\text{std}\) or \(-qlanglvl=95\text{std}\), this
extension is disabled in all compile-time format specifications, and the form of \( \theta X \) is enforced. To disable this extension in run-time formats, the following run-time option must be set:

\[
\text{XLFRTEOPTS="langlvl=90std" or "langlvl=95std" ; export XLFRTEOPTS}
\]

---
End of IBM Extension
---

**Examples**

**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')

**List-directed formatting**

List-directed formatting allows you to control the editing process using the lengths and types of data that is read or written. You can only use list-directed formatting with sequential or stream access.

Use the asterisk format identifier to specify list-directed formatting. For example:

```
REAL TOTAL1, TOTAL2
PRINT *, TOTAL1, TOTAL2
```

**Value separators**

If you specify list-directed formatting for a formatted record, that record consists of a sequence of values and value separators.

where:

- **value** is a constant or null.
- **value separator** is a comma, slash, or set of adjacent blanks that occur between values in a record. You can specify one or more blanks before and after a comma or slash.
- **null** is 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.

A null value has no effect on the definition status of the corresponding input list item.

**List-directed input**

Input list items in a list-directed `READ` statement are defined by corresponding values in a formatted record. The syntax of each value must agree with the type of the corresponding input list item.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Type</th>
</tr>
</thead>
</table>

---

Chapter 9. Input/Output formatting 223
Table 24. List-directed input (continued)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$c$</td>
<td>A literal constant of intrinsic type, or a non-delimited character constant.</td>
</tr>
<tr>
<td>$r^*$</td>
<td>$r$ is an unsigned, nonzero, integer literal constant. $r^*$ indicates $r$ successive appearances of the null value.</td>
</tr>
<tr>
<td>$r^*c$</td>
<td>Indicates $r$ successive appearances of the constant.</td>
</tr>
</tbody>
</table>

**Rules for list-directed input**

You must not specify a kind type parameter for $c$ or $r$.

List-directed formatting interprets two or more consecutive blanks as a single blank, unless the blanks are within a character value.

The constant $c$ will have the same kind type parameter as the corresponding list item.

---

**IBM Extension**

Use the `-qintlog` compiler option to specify integer or logical values for input items of either integer or logical type.

---

**End of IBM Extension**

List-directed formatting interprets an object of derived type that occurs in an input list as if all structure components occur in the same order as in the derived type definition. The ultimate components of the derived type must not have the pointer or allocatable attribute.

A slash indicates the end of the input list and terminates list-directed formatting. Additional input list items after the slash evaluate as null values.

**Continuing a character value**

A character value that meets the following conditions can continue in as many records as necessary:
- The next item or ultimate component of a derived type is of type character.
- The character constant does not contain the value separators blank, comma, or slash.
- The character constant does not cross a record boundary.
- The first non-blank character is not a quotation mark or apostrophe.
- The leading characters are non numeric and followed by an asterisk.
- The character constant contains at least one character.

Delimiting apostrophes or quotation marks are not necessary to continue a character value across multiple records. If you omit delimiting characters, the first blank, comma, slash, or end-of-record terminates the character constant.

If you do not specify delimiting apostrophes or quotation marks, apostrophes and double quotation marks in the character value are not doubled.

**End-of-record and list-directed input**

In list-directed input an end-of-record has the same effect as a blank separator, unless the blank is within a character literal constant or complex literal constant. An end-of-record does not insert a blank or any other character in a character
value. An end-of-record must not occur between a doubled apostrophe in an apostrophe-delimited character sequence, or between a doubled quote in a quote-delimited character sequence.

**List-directed output**

List-directed **PRINT** and **WRITE** statements output values in an order identical to the output list. Values are written in a form valid for the data type of each output list item.

**Types of list-directed output**

*Table 25. List-directed output*

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Form of Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrays</td>
<td>Column-major order</td>
</tr>
<tr>
<td>Character</td>
<td>Depends on DELIM= specifier and file type, see Character Output</td>
</tr>
<tr>
<td>Complex</td>
<td>Enclosed in parentheses with a comma separating the real and imaginary parts. Uses E or F editing.</td>
</tr>
<tr>
<td>Integer</td>
<td>Uses I editing.</td>
</tr>
<tr>
<td>Logical</td>
<td>T for a true value F for a false value</td>
</tr>
<tr>
<td>Real</td>
<td>Uses E or F editing.</td>
</tr>
</tbody>
</table>

**List-directed character output**

The output of character constants can change depending on the **DELIM=** specifier on the **OPEN** or **READ** statements.

Character constants output to a file opened without a **DELIM=** specifier, or a file opened with a **DELIM=** specifier with a value of NONE, output as follows:

- Values are not delimited by apostrophes or quotation marks.
- **Value separators** do not occur between values.
- Each internal apostrophe or double quotation mark outputs as one apostrophe or double quotation mark.
- The processor inserts a blank character for carriage control at the beginning of any record that continues a character constant from the preceding record.

**Note:** Non-delimited character data can not always be read back correctly using list-directed input. Use with discretion.

Double quotation marks delimit character constants in a file opened with a **DELIM=** specifier with a value of QUOTE. A value separator follows the delimiter. Each internal quote outputs as two contiguous double quotation marks.

Apostrophes delimit character constants in a file opened with a **DELIM=** specifier with a value of APOSTROPHE. A value separator follows the delimiter. Each internal apostrophe outputs as two contiguous apostrophes.

**Rules for list-directed output**

Each output record begins with a blank character that provides carriage control when that record outputs.
The end-of-record must not occur within a constant that is not character or complex.

In a complex constant, the end of a record can occur between the comma and the imaginary part of the constant only if the constant is as long or longer than a record. The only embedded blanks that can occur 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.

Blanks must not occur within a constant that is not character or complex.

Null values are not output.

Slashes you specify as value separators are not output.

---

**IBM Extension**

The *Width of a Written Field* table contains the width of the written field for any data type and length. The size of the record is the sum of the field widths plus one byte to separate each non-character field.

*Table 26. Width of a written field*

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Length (bytes)</th>
<th>Maximum Field Width (characters)</th>
<th>Fraction (decimal digits)</th>
<th>Precision/IEEE (decimal digits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>1</td>
<td>4</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>6</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>11</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>20</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>real</td>
<td>4</td>
<td>17</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>26</td>
<td>18</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>43</td>
<td>35</td>
<td>31</td>
</tr>
<tr>
<td>complex</td>
<td>8</td>
<td>37</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>55</td>
<td>18</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>89</td>
<td>35</td>
<td>31</td>
</tr>
<tr>
<td>logical</td>
<td>1</td>
<td>1</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>character</td>
<td>n</td>
<td>n</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

---

**Namelist formatting**

Namelist formatting allows you to use the NAME= specifier as part of the NAMELIST statement to assign a name to a collection of variables. This name represents the entire collection of variables for input and output. You can also use namelist formatting to include namelist comments with input, making the data more user accessible.

- In Fortran 90 and Fortran 95, you can only use namelist formatting with sequential access.
- The Fortran 2003 Standard allows you to use namelist formatting with sequential and stream access.
The Fortran 2003 Standard allows you to use namelist formatting with internal files.

**Namelist input**

The form of namelist input is:

1. Optional blanks and namelist comments.
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.

---

**IBM Extension**

If you specify the `NAMELIST=OLD` run-time option, the form of input for a `NAMELIST` statement is:

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 by a single comma. You can insert a comma after the last name-value subsequence.
5. `&END` or `$END` to terminate the namelist input.

---

**End of IBM Extension**

The first character of each input record must be a blank, including those records that continue a delimited character constant.

**Namelist comments**

---

**Fortran 95**

In Fortran 95 and higher, you can use comments in namelists.

---

**Fortran 2003 Standard**

You must not specify comments in stream input.

---

**End of Fortran 2003 Standard**

If you specify the `NAMELIST=NEW` run-time option:

- If you specify an exclamation point after a value separator that is not a slash, or in the first non-blank position of a namelist input record, you initiate a comment. You can not initiate comments inside character literal constants.
- The comment extends to the end of the input record, and can contain any character in the XL Fortran character set.
- The comment is ignored.
A slash within a namelist comment does not terminate execution of that namelist input statement.

--- IBM Extension ---

If you specify the **NAMELIST=OLD** run-time option:

- If you specify an exclamation point after a single comma or in the first non-blank position of a namelist input record that is not the first character of that record, you initiate a comment. You must not initiate a namelist comment within a character literal constant.
- The comment extends to the end of the input record, and can contain any character in the **XL Fortran character set**.
- The comment is ignored.
- An &END or $END within a namelist comment does not terminate execution of the namelist input statement.

--- End of IBM Extension ---

--- End of Fortran 95 ---

**Name-value subsequence**

The form of a name-value subsequence in an input record is:

```
name = constant_list
```

- `name` is a variable
- `constant` has the following forms:

```
r * literal_constant
```

- `r` is an unsigned, nonzero, scalar, integer literal constant specifying the number of times the `literal_constant` occurs. You must not specify a kind type parameter for `r`.
- `literal_constant` is a scalar literal constant of intrinsic type, or null value. You must not specify a kind type parameter for the constant. The constant evaluates with the same kind type parameter as the corresponding list item.

You must specify delimiting apostrophes or quotation marks if `literal_constant` is of type character.

You can specify T or F if `literal_constant` is of type logical.
Rules for namelist input

Any subscripts, strides, and substring range expressions that qualify name must be integer literal constants with no kind type parameter.

If name is not an array or an object of derived type constant_list must contain a single constant.

Variable names you specify in the input file must appear in the variable_name_list of a NAMELIST statement. Variables can appear in any order.

If a name that you specify in an EQUIVALENCE statement shares storage with name, you must not substitute for that name in the variable_name_list.

You can use one or more optional blanks before or after name, but name must not contain embedded blanks.

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.
- zero-length character string.

If you specify the optional qualification, it must not contain a vector subscript.

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.

You can specify a null value using:

- The r* form that indicates r successive appearances of the null value.
- 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 non-blank 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.

You must not use a null value as the real or imaginary part of a complex constant. 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.

---

IBM Extension

When you set the LANGLVL=EXTENDED run-time option, XL Fortran allows you to specify multiple input values in conjunction with a single array element. XL Fortran assigns the values to successive elements of that array, in array element order. The array element must not specify subobject designators.
Consider the following example, which declares array A as follows:

```
INTEGER A(100)
NAMELIST /FOO/ A
READ (5, FOO)
```

Unit 5 contains the following input:

```
&FOO
A(3) = 2, 10, 15, 16
/
```

During execution of the `READ` statement, XL Fortran assigns the following values:

- 2 to A(3)
- 10 to A(4)
- 15 to A(5)
- 16 to A(6)

If you specify multiple values in conjunction with a single array element, any logical constant must be specified with a leading period, for example, `.T`.

If you use the `NAMELIST=OLD` option at run-time, the `BLANK=` specifier in the `OPEN` or `READ` statements determines how XL Fortran interprets embedded and trailing blanks between non-character constants.

If you specify the `-qmixed` compiler option, the namelist group name and list item names are case-sensitive.

---

A slash appearing as a value separator terminates the input statement after assignment of the previous value. Any additional items in the namelist group object receive null values

**Example of namelist input data**

File `NMLEXP` contains the following data before execution of the `READ` statement.

Character position:

```
1 2 3
1....0....0....0....0
```

File contents:

```
&NAME1
I=5,
SMITH&P_AGE=27
/
```

`NMLEXP` 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')
```
Note: In the previous example, data items appear in separate data records. The next example is a file with the same data items in one data record:

Character position:

1 2 3 4
1...0...0...0...0...0...

File contents:

&NAME1 I= 5, SMITH%P_AGE=40 /

--- Fortran 95 ---

An example of a [NAMELIST comment] when you specify [NAMELIST=NEW]. The 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)/

--- End of Fortran 95 ---

--- IBM Extension ---

An example of a [NAMELIST comment] when you specify [NAMELIST=OLD]. The 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) &END

--- End of IBM Extension ---

**Namelist output**

The [WRITE] statement outputs data from the variable name list in a [NAMELIST] statement according to data type. This data can be read using [namelist input] except for non-delimited character data.

You must not specify a single long character variable for namelist output.

Each output record that is not continuing a delimited character constant from a previous record begins with a blank character that provides carriage control.

The output data fields become large enough to contain all significant digits, as shown in the [Width of a Written Field] table.

The values of a complete array output in column-major order.

If the length of an array element is not sufficient to hold the data, you must specify an array with more than three elements.
A `WRITE` statement with a `variable_name_list` produces a minimum of three output records:

- One record containing the namelist name.
- One or more records containing the output data items.
- One record containing a slash to terminate output.

To output namelist data to an internal file, the file must be a character array containing at least three elements. If you use the `WRITE` statement to transfer data to an internal file, the character array can require more than three elements.

---

You can delimit character data using the `DELIM=` specifier on the `OPEN` or `READ` statements.

**Namelist character output**

The output of character constants can change depending on the `DELIM=` specifier on the `OPEN` or `READ` statements.

For character constants in a file opened without a `DELIM=` specifier, or with a `DELIM=NONE`:

- Values are non-delimited by apostrophes or quotation marks.
- Value separators do not occur between values.
- Each internal apostrophe or double quotation mark outputs as one apostrophe or quotation mark.
- XL Fortran inserts a blank character for carriage control at the beginning of any record that continues a character constant from the preceding record.

Nondelimited character data that has been written must not be read as character data.

Double quotation marks delimit character constants in a file opened with `DELIM=QUOTE` with a value separator preceding and following each constant. Each internal quote outputs as two contiguous quotation marks.

Apostrophes delimit character constants in a file opened with `DELIM=APOSTROPHE` with a value separator preceding and following each constant. Each internal apostrophe outputs as two contiguous apostrophes.

**Rules for namelist output**

You must not specify a single character variable to output namelist data to an internal file, even if it is large enough to hold all of the data.

If you do not specify the `NAMELIST` run-time option, or you specify `NAMELIST=NEW` the namelist group name and namelist item names output in uppercase.

---

If you specify `NAMELIST=OLD` at run-time:

- The namelist group name and namelist item names output in lower case.
- An `&END` terminates the output record.
If you specify `NAMELIST=OLD` at run-time and do not use the `DELIM=` specifier on an `OPEN` or `READ` statement:

- Apostrophes delimit character data
- Apostrophes delimit non-delimited character strings. A comma separator occurs between each character string.
- If a record starts with the continuation of a character string from the previous record, blanks are not added to the beginning of that record.

If you use the `-qmixed` compiler option, the namelist group name is case sensitive, regardless of the value of the `NAMELIST` run-time option.

To restrict namelist output records to a given width, use the `RECL=` specifier on the `OPEN` statement, or the `NLWIDTH` run-time option.

By default all output items for external files appear in a single output record. To have the record output on separate lines, use the `RECL=` specifier on the `OPEN` statement, or the `NLWIDTH` run-time option.

---

**Example of namelist output data**

```fortran
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
&end
```

---

**IBM Extension**

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
```

---

End of IBM Extension
Chapter 10. Statements and attributes

This section 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.

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.

Notes:
1. IBM Extension.
2. Fortran 95.

Table 27. Statements table

<table>
<thead>
<tr>
<th>Statement Name</th>
<th>Executable Statement</th>
<th>Specification Statement</th>
<th>Terminal Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALLOCATABLE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALLOCATE</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>ASSIGN</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASSOCIATE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AUTOMATIC</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>BLOCK DATA</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BYTE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALL</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>CASE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHARACTER</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMMON</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>COMPLEX</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONTAINS</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONTINUE</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>CYCLE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATA</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEALLOCATE</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Derived Type</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIMENSION</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DO</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DO WHILE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOUBLE COMPLEX</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Statement Name</td>
<td>Executable Statement</td>
<td>Specification Statement</td>
<td>Terminal Statement</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------------</td>
<td>-------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRECISION</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ELSE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ELSE IF</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ELSEWHERE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END ASSOCIATE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END BLOCK DATA</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END DO</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END ENUM</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END IF</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END FORALL</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END FUNCTION</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END INTERFACE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END MAP</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END MODULE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END PROGRAM</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END SELECT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END SUBROUTINE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END STRUCTURE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END TYPE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END UNION</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END WHERE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENDFILE</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>ENTRY</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENUM</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENUMERATOR</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EQUIVALENCE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXIT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXTERNAL</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLUSH</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FORALL</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FORMAT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FUNCTION</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GO TO (Assigned)</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GO TO (Computed)</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>GO TO (Unconditional)</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Statement Name</td>
<td>Executable Statement</td>
<td>Specification Statement</td>
<td>Terminal Statement</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------------------</td>
<td>-------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>IF (Block)</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IF (Arithmetic)</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IF (Logical)</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMPLICIT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMPORT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INQUIRE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTENT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTERFACE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTRINSIC</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOGICAL</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAP</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MODULE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MODULE PROCEDURE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NAMELIST</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NULLIFY</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPEN</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPTIONAL</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARAMETER</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PAUSE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POINTER (Fortran 90)</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POINTER (integer)</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRINT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRIVATE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROCEDURE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROGRAM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROTECTED</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUBLIC</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>READ</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RECORD</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RETURN</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REWind</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAVE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SELECT CASE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEQUENCE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Statement Function</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 27. Statements table (continued)

<table>
<thead>
<tr>
<th>Statement Name</th>
<th>Executable Statement</th>
<th>Specification Statement</th>
<th>Terminal Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATIC</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>STOP</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SUBROUTINE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STRUCTURE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>TARGET</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>TYPE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Type Declaration</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>UNION</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>USE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>VALUE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>VECTOR</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>VIRTUAL</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>VOLATILE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>WAIT</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>WHERE</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>WRITE</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Assignment and pointer assignment statements are discussed in [Chapter 5, “Expressions and assignment,” on page 83](#). 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:

```fortran
REAL, PRIVATE :: A  ! Type declaration statement
PRIVATE :: A       ! Attribute specification statement

MODULE X
PRIVATE
REAL :: A
END MODULE
```

### ALLOCATABLE

#### Purpose

The ALLOCATABLE attribute declares allocatable objects—that is, objects whose space is dynamically allocated by execution of an ALLOCATE statement or by a derived-type assignment statement. If it is an array, it will be a deferred-shape array.
**Syntax**

```
object_name is the name of an allocatable object
deferred_shape_spec is a colon(¦), where each colon represents a dimension
```

**Rules**

The object cannot be a pointee. If the object is an array and it is specified elsewhere in the scoping unit with the `DIMENSION` attribute, the array specification must be a `deferred_shape_spec`.

**Table 28. Attributes compatible with the ALLOCATABLE attribute**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTOMATIC</td>
<td>STATIC</td>
</tr>
<tr>
<td>DIMENSION</td>
<td>TARGET</td>
</tr>
<tr>
<td>INTENT</td>
<td>VOLATILE</td>
</tr>
<tr>
<td>OPTIONAL</td>
<td>PRIVATE</td>
</tr>
<tr>
<td></td>
<td>PROTECTED</td>
</tr>
<tr>
<td></td>
<td>PUBLIC</td>
</tr>
<tr>
<td></td>
<td>SAVE</td>
</tr>
</tbody>
</table>

**Examples**

```fortran
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 69
- "ALLOCATED(X)" on page 485
- "ALLOCATE" on page 240
- "DEALLOCATE" on page 276
- "Allocation status" on page 58
- "Deferred-shape arrays" on page 68
- "Allocatable objects as dummy arguments" on page 166
- "Allocatable components" on page 36
ALLOCATE

Purpose

The ALLOCATE statement dynamically provides storage for pointer targets and allocatable objects.

Syntax

\[
\text{ALLOCATE}(\text{allocation_list}, \text{STAT} = \text{stat_variable})
\]

\textit{stat_variable}

is a scalar integer variable

\textit{allocation_list}

\[
\text{allocate_object}, (\text{lower_bound} : \text{upper_bound})
\]

\textit{allocate_object}

is a variable name or structure component. It must be a data pointer or an allocatable object.

\textit{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 object, the object becomes definable.

The number of dimensions specified (i.e., the number of upper bounds in \textit{allocate_object}) must be equal to the rank of \textit{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 \textit{allocate_object} is zero-sized.

Fortran 2003 Standard

Any \textit{allocate_object} or a specified bound of an \textit{allocate_object} must not depend on \textit{stat_variable} or on the value, bounds, allocation status, or association status of any \textit{allocate_object} in the same ALLOCATE statement.
Stat_variable must not be allocated within the ALLOCATE statement in which they appear. They also must not depend on the value, bounds, length type parameters, allocation status, or association status of any allocate_object in the same ALLOCATE statement.

End of Fortran 2003 Standard

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:

<table>
<thead>
<tr>
<th>Stat value</th>
<th>Error condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No error</td>
</tr>
<tr>
<td>1</td>
<td>Error in system routine attempting to do allocation</td>
</tr>
<tr>
<td>2</td>
<td>An invalid data object has been specified for allocation</td>
</tr>
<tr>
<td>3</td>
<td>Both error conditions 1 and 2 have occurred</td>
</tr>
</tbody>
</table>

IBM Extension

Allocating an allocatable object 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

When an object of derived type is created by an ALLOCATE statement, any allocatable ultimate components have an allocation status of not currently allocated.

Use the ALLOCATED intrinsic function to determine if an allocatable object 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.ALOCATED(A)) ALLOCATE (A(N**2))
END
Related information

- “ALLOCATE” on page 238
- “DEALLOCATE” on page 276
- “Allocation status” on page 58
- “Pointer association” on page 136
- “Deferred-shape arrays” on page 68
- “ALLOCATED(X)” on page 485
- “ASSOCIATED(POINTER, TARGET)” on page 488
- “Allocatable objects as dummy arguments” on page 166
- “Allocatable components” on page 36

ASSIGN

Purpose

The ASSIGN statement assigns a statement label to an integer variable.

Syntax

```
--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.
Fortran 95

The ASSIGN statement has been deleted from Fortran 95 and higher.

End of 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 5
- “GO TO (assigned)” on page 321
- “Deleted features” on page 746

ASSOCIATE

Fortran 2003 Standard

Purpose

An ASSOCIATE statement is the first statement in an ASSOCIATE construct. It establishes an association between each identifier and a variable, or the value of an expression.

Syntax

```
associate_construct_name:ASSOCIATE(associate_name=>selector)
```

- `associate_construct_name` is a name that identifies the ASSOCIATE construct.
- `associate_name` is an identifier that once associated with the selector, becomes an associating entity.
- `selector` is a variable or expression that once associated becomes the associated entity.

Rules

If the selector is an expression or a variable with a vector subscript, the `associate_name` is assigned the value of the expression or variable. That associating entity must not become redefined or undefined.
If the selector is a variable without a vector subscript, the associate_name is associated with the data object specified by the selector. Whenever the value of the associate_name (or the associating entity identified by the associate_name) changes, the value of the variable changes with it.

If the selector has the ALLOCATABLE attribute, the associating entity does not have the ALLOCATABLE attribute. If the selector has the POINTER attribute, then the associating entity has the TARGET attribute. If the selector has the TARGET or VOLATILE attribute, the associating entity that is a variable has those attributes.

If the selector has the OPTIONAL attribute, it must be present.

An associating entity has the same type, type parameters, and rank as the selector. If the selector is an array, the associating entity is an array with a lower bound for each dimension equal to the value of the intrinsic LBOUND(selector). The upper bound for each dimension is equal to the lower bound plus the extents minus 1.

An associate_name must be unique within an ASSOCIATE construct.

If the associate_construct_name appears on an ASSOCIATE construct statement, it must also appear on the corresponding END ASSOCIATE statement.

An ASSOCIATE construct statement must not appear within the dynamic or lexical extent of a parallel region.

Examples

```fortran
test_equiv: ASSOCIATE (a1 => 2, a2 => 40, a3 => 80)  
  IF ((a1 * a2) .eq. a3) THEN  
    PRINT *, "a3 = (a1 * a3)"  
  END IF  
END ASSOCIATE test_equiv
END
```

Related information

The scope of a name

End of Fortran 2003 Standard

AUTOMATIC

IBM Extension

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.

Syntax

```
AUTOMATIC automatic_list
```
**automatic**

is a variable name or an array declarator with an explicit-shape specification list or a deferred-shape specification list

**Rules**

If `automatic` is a function result 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.

A variable with the AUTOMATIC attribute cannot be initialized by a DATA statement or a type declaration statement.

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.

**Note:** An object with the AUTOMATIC attribute should not be confused with an automatic object. See “Automatic objects” on page 16.

---

**Attributes compatible with the AUTOMATIC attribute**

- ALLOCATABLE
- DIMENSION
- POINTER
- TARGET
- VOLATILE

**Examples**

```fortran
CALL SUB
CONTAINS
    SUBROUTINE SUB
        INTEGER, AUTOMATIC :: VAR
        VAR = 12
    END SUBROUTINE
END
```

**Related information**

- “Storage classes for variables” on page 59
- `-qinitauto` option in the `XL Fortran Compiler Reference`

---

End of IBM Extension
### BACKSPACE

#### Purpose

The BACKSPACE statement positions an external file connected for sequential access or formatted stream access.

#### Syntax

```
BACKSPACE u (position_list)
```

- `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 an 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.
  - `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.

- `IOMSG= iomsg_variable` is an input/output status specifier that specifies the message returned by the input/output operation. `iomsg_variable` is a scalar default character variable. It must not be a use-associated nonpointer protected variable. When the input/output statement containing this specifier finishes execution, `iomsg_variable` is defined as follows:
  - If an error, end-of-file, or end-of-record condition occurs, the variable is assigned an explanatory message as if by assignment.
  - If no such condition occurs, the value of the variable is unchanged.

#### Fortran 2003 Standard

- `IOSTAT= ios` is an input/output status specifier that specifies the status of the input/output operation. `ios` is an variable. 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.
Rules

After the execution of a BACKSPACE statement, the file position is before the current record if a current record exists. If there is no current record, the file position is before the preceding record. If the file is at its initial point, file position remains unchanged.

You cannot backspace over records that were written using list-directed or namelist formatting.

For sequential access, if the preceding record is the endfile record, the file is positioned before the endfile record.

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.

IBM Extension

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.

End of IBM Extension

Examples

BACKSPACE 15
BACKSPACE (UNIT=15,ERR=99)
...
99 PRINT *, "Unable to backspace file."
END

Related information

• “Conditions and IOSTAT values” on page 186
• Chapter 8, “XL Fortran Input/Output,” on page 177
• Setting run-time options in the XL Fortran Compiler Reference

BIND

Fortran 2003 Standard

Purpose

The BIND attribute declares that a Fortran variable or common block is interoperable with the C programming language.

Syntax
binding_label

is a scalar character initialization expression

Rules

This attribute specifies that a Fortran variable or common block is interoperable with a C entity with external linkage. Refer to "Interoperability of Variables" on page 680 and "Interoperability of common blocks" on page 680 for details.

If the NAME= specifier appears in a BIND statement, then only one variable_name or common_block_name can appear.

If a BIND statement specifies a common block, then each variable of that common block must be of interoperable type and type parameters, and must not have the POINTER or ALLOCATABLE attribute.

Table 29. Attributes compatible with the BIND attribute

<table>
<thead>
<tr>
<th>DIMENSION</th>
<th>SAVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRIVATE</td>
<td>STATIC</td>
</tr>
<tr>
<td>PROTECTED</td>
<td>TARGET</td>
</tr>
<tr>
<td>PUBLIC</td>
<td>VOLATILE</td>
</tr>
</tbody>
</table>

Related information

- Chapter 16, "Language interoperability features," on page 679
- "Interoperability of Variables" on page 680
- "Interoperability of common blocks" on page 680
- "ENTRY" on page 300
- "FUNCTION" on page 317
- "PROCEDURE" on page 371
- "SUBROUTINE" on page 401
- "Derived Type" on page 278

---

End of Fortran 2003 Standard

---

**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.

**Syntax**
**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**

```plaintext
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 153
- “END” on page 292 for details on the END BLOCK DATA statement

**BYTE**

IBM Extension

**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.

**Syntax**

```plaintext
 BYTE entity_decl_list::,
 attr_spec_list::
```
where:

<table>
<thead>
<tr>
<th>attr_spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALLOCATABLE</td>
</tr>
<tr>
<td>AUTOMATIC</td>
</tr>
<tr>
<td>BIND</td>
</tr>
<tr>
<td>DIMENSION (array_spec)</td>
</tr>
<tr>
<td>EXTERNAL</td>
</tr>
<tr>
<td>INTENT (intent_spec)</td>
</tr>
<tr>
<td>INTRINSIC</td>
</tr>
<tr>
<td>OPTIONAL</td>
</tr>
<tr>
<td>PARAMETER</td>
</tr>
<tr>
<td>POINTER</td>
</tr>
<tr>
<td>PRIVATE</td>
</tr>
<tr>
<td>PUBLIC</td>
</tr>
<tr>
<td>SAVE</td>
</tr>
<tr>
<td>STATIC</td>
</tr>
<tr>
<td>TARGET</td>
</tr>
<tr>
<td>VOLATILE</td>
</tr>
</tbody>
</table>

**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. Use the double colon separator when you specify attributes, =initialization_expr, or => NULL().

**array_spec**

is a list of dimension bounds

**entity_decl**

```
(a (array_spec) initial_value_list = initialization_expr => NULL())
```

**a** is an object name or function name. **array_spec** cannot be specified for a function with an implicit interface.

**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 object, 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 `initialization_expr` is specified, and the entity you are declaring:

• is a variable, the variable is initially defined.

```
Fortran 95
```

• is a derived type component, the derived type has default initialization.

```
End of Fortran 95
```

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 \( \text{initialization} \_\text{expr} \) or \( \text{NULL}() \) 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\_\text{spec} \text{specified in the } \text{entity\_decl} \text{ takes precedence over the } \array\_\text{spec} \text{ in the } \text{DIMENSION} \text{ attribute.}

An array function result that does not have the \( \text{ALLOCATABLE} \) or \( \text{POINTER} \) 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**

\[
\text{BYTE, DIMENSION(4) :: X=(/1,2,3,4/)}
\]

**Related information**

- “BYTE” on page 28
- “Initialization expressions” on page 85
- “How type is determined” on page 51, for details on the implicit typing rules
- “Automatic objects” on page 16
- “Storage classes for variables” on page 59
- “DATA” on page 273, for details on initial values

---

**CALL**

**Purpose**

The \text{CALL} \text{statement} \text{invokes} \text{a subroutine} \text{to be executed.}

**Syntax**

\[
\text{name}(\text{actual\_argument\_spec\_list})
\]

\text{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 \text{CALL} \text{statement} \text{results} \text{in the following order} \text{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 \texttt{RECURSIVE} keyword.

If a \texttt{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 \texttt{RETURN} statement.

---

\textbf{IBM Extension}

An external subprogram can also refer to itself directly or indirectly if the \texttt{-qrecur} compiler option is specified.

The argument list built-in functions \texttt{%VAL} and \texttt{%REF} are supplied to aid interlanguage calls by allowing arguments to be passed by value and by reference, respectively. They can only be references to non-Fortran procedures

---

\textbf{Fortran 2003 Standard}

The \texttt{VALUE} attribute also allows you to pass arguments by value.

---

\textbf{Examples}

\begin{verbatim}
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
\end{verbatim}

\textbf{Related information}

- “Recursion” on page 171
- “%VAL and %REF” on page 161
- “VALUE” on page 417
- “Actual argument specification” on page 157
- “Asterisks as dummy arguments” on page 168
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.

Syntax

```
CASE case_selector case_construct_name
```

case_selector

```
DEFAULT
```

```
(case_value
  low_case_value : high_case_value)
```

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 -qctypless 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

- “SELECT CASE construct” on page 125
- “SELECT CASE” on page 395
- “END (Construct)” on page 293, 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.

Syntax

where:

\[
\text{attr_spec} \quad \text{ALLOCATABLE, AUTOMATIC, BIND, DIMENSION (array_spec), EXTERNAL, INTENT (intent_spec), INTRINSIC, OPTIONAL, PARAMETER, POINTER, PRIVATE, PUBLIC, SAVE, STATIC, TARGET, VOLATILE}
\]

\[\text{char_selector} \quad \text{specifies the character length.}\]

\[
\begin{align*}
\text{LEN} &= \text{type_param_value} , \quad \text{KIND} = \text{int_init_expr} \\
\text{KIND} &= \text{int_init_expr} , \quad \text{LEN} = \text{type_param_value} \\
\text{LEN} &= \text{char_length} \\
\end{align*}
\]
*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. Use the double colon separator when you specify attributes, :=*initialization_expr*, or => NULL().

*array_spec*

is a list of dimension bounds.

*entity_decl*

```
[ /*char_length*/ ]
[ /*array_spec*/ ]
[ /*array_spec*/ ] /*char_length*/
```

```
[/*initial_value_list*/]
=/*initialization_expr*/
(1)

=/>/*initialization_expr*/
(2)
```

Notes:

1 IBM Extension

2 Fortran 95

*a* is an object name or function name. *array_spec* cannot be specified for a function with an implicit interface.

```
IBM Extension
```

*initial_value*

provides an initial value for the entity specified by the immediately preceding name

```
End of IBM Extension
```

*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 must not be initially defined in a type declaration statement if it is a dummy argument, an allocatable object, 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.

- 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
The specification expression of a \texttt{type\_param\_value} or an \texttt{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 \texttt{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.

\textit{initialization\_expr} must be specified if the statement contains the \texttt{PARAMETER} attribute. If \textit{initialization\_expr} or \texttt{NULL()} is specified, and the entity you are declaring:

- is a variable, the variable is initially defined.

\texttt{a} becomes defined with the value determined by \textit{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 \texttt{a} is a variable, the presence of \textit{initialization\_expr} or \texttt{NULL()} implies that \texttt{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 \texttt{array\_spec} specified in an \texttt{entity\_decl} takes precedence over the \texttt{array\_spec} in the \texttt{DIMENSION} attribute. A \textit{char\_length} specified in an \texttt{entity\_decl} takes precedence over any length specified in \texttt{char\_selector}.

An array function result that does not have the \texttt{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.

\begin{itemize}
  \item \texttt{T} or \texttt{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.
\end{itemize}

The optional comma after \textit{char\_length} in a \texttt{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**

```fortran
CHARACTER(KIND=1,LEN=6) APPLES /'APPLES'/
CHARACTER(7), TARGET :: ORANGES = 'ORANGES'
I=7
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 25
- “Initialization expressions” on page 85
- “How type is determined” on page 51 for details on the implicit typing rules
- “Array declarators” on page 65
- “Automatic objects” on page 16
- “Storage classes for variables” on page 59
- “DATA” on page 273 for details on initial values

---

**CLOSE**

**Purpose**

The CLOSE statement disconnects an external file from a unit.

**Syntax**
CLOSE—(close_list)—

close_list
is a list that must contain one unit specifier (UNIT=μ) and can also contain one of each of the other valid specifiers. The valid specifiers are:

[UNIT=] μ
is a unit specifier in which μ 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 an integer expression, whose value is in the range 1 through 2147483647. If the optional characters UNIT= are omitted, μ must be the first item in close_list.

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.

Fortran 2003 Standard

IOMSG= iomsg_variable
is an input/output status specifier that specifies the message returned by the input/output operation. iomsg_variable is a scalar default character variable. It must not be a use-associated nonpointer protected variable. When the input/output statement containing this specifier finishes execution, iomsg_variable is defined as follows:
- If an error, end-of-file, or end-of-record condition occurs, the variable is assigned an explanatory message as if by assignment.
- If no such condition occurs, the value of the variable is unchanged.

End of Fortran 2003 Standard

IOSTAT= ios
is an input/output status specifier that specifies the status of the input/output operation. ios is an integer variable. 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.

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.

IBM 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

- "Units" on page 180
- "Conditions and IOSTAT values" on page 186
- "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.

Syntax

```
COMMON object_list
    common_block_name/object_list
```

object
Rules

object cannot refer to a dummy argument, automatic object, allocatable object, or an object of a derived type that has an allocatable ultimate component, 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.

A given variable_name or procedure pointer name can only appear once in all common block object lists within a scoping unit. Their names cannot be made accessible by use association.

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.

A procedure pointer can be storage associated only with another procedure pointer; both interfaces must be either explicit or implicit. If both interfaces are explicit, their characteristics must be the same. If both interfaces are implicit, both must be subroutines or both must be functions with the same type and type parameters.
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.

---

**Fortran 2003 Standard**

If a common block has the `BIND` attribute, it must have the `BIND` attribute and the same binding label in each scoping unit in which it is declared. A C variable with external linkage interoperates with a common block with the `BIND` attribute if:

- The C variable is of a struct type and the variables that are members of the common block are interoperable with corresponding components of the struct type, or
- The common block contains a single variable, and the variable is interoperable with the C variable.

---

**IBM Extension**

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 [Pthreads library module](#) in the `XL Fortran Optimization and Programming Guide` for more information. The `lock_name` attribute on the `CRITICAL` directive also provides the ability to serialize access to data. See `CRITICAL / END CRITICAL` in the `XL Fortran Optimization and Programming Guide` for more information. The `THREADLOCAL` and `THREADPRIVATE` directives ensure that common blocks are local to each thread. See `THREADLOCAL` and `THREADPRIVATE` in the `XL Fortran Optimization and Programming Guide` 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:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>A</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable A:</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable B:</td>
<td></td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Array C:</td>
<td></td>
<td></td>
<td>C(1)</td>
<td></td>
<td>C(2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 **DATA** program unit containing a **DATA** statement or a type declaration statement. You cannot initially define any elements of 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

- Pthreads library module in the *XL Fortran Optimization and Programming Guide*
- “BIND” on page 247
- THREADLOCAL
- “Block data program unit” on page 153
- “Explicit-shape arrays” on page 66
- “The scope of a name” on page 130 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.

**Syntax**

```
COMPLEX(kind_selector) :: entity_decl_list
                   ,attr_spec_list::
```
where:

<table>
<thead>
<tr>
<th>attr_spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALLOCATABLE</td>
</tr>
<tr>
<td>AUTOMATIC</td>
</tr>
<tr>
<td>BIND</td>
</tr>
<tr>
<td>DIMENSION (array_spec)</td>
</tr>
<tr>
<td>EXTERNAL</td>
</tr>
<tr>
<td>INTENT (intent_spec)</td>
</tr>
<tr>
<td>INTRINSIC</td>
</tr>
<tr>
<td>OPTIONAL</td>
</tr>
<tr>
<td>PARAMETER</td>
</tr>
<tr>
<td>POINTER</td>
</tr>
<tr>
<td>PRIVATE</td>
</tr>
<tr>
<td>PUBLIC</td>
</tr>
<tr>
<td>SAVE</td>
</tr>
<tr>
<td>STATIC</td>
</tr>
<tr>
<td>TARGET</td>
</tr>
<tr>
<td>VOLATILE</td>
</tr>
</tbody>
</table>


kind_selector

```
( int_initialization_expr )
```

Notes:
1 IBM Extension.

specifies the length of complex entities:

--- IBM Extension

- 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.

--- End of IBM Extension

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. Use the double colon separator when you specify attributes, `=initialization_expr`, or `=NULL()`.

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 with an implicit interface.

IBM Extension

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.

_End of IBM Extension_

IBM Extension

initial_value provides an initial value for the entity specified by the immediately preceding name

_End of IBM Extension_

initialization_expr provides an initial value, by means of an initialization expression, for the entity specified by the immediately preceding name

Fortran 95

=> NULL() provides an initial value for the pointer object

_End of Fortran 95_
Within the context of a derived type definition:

- If => appears in a component initialization, the \texttt{POINTER} attribute must appear in the \textit{attr_spec_list}.
- If = appears in a component initialization, the \texttt{POINTER} attribute cannot appear in the component \textit{attr_spec_list}.
- The compiler will evaluate \textit{initialization_expr} within the scoping unit of the type definition.

If => appears for a variable, the object must have the \texttt{POINTER} attribute.

If \textit{initialization_expr} appears for a variable, the object cannot have the \texttt{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 object, 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 \texttt{AUTOMATIC} attribute. The object may be initialized if:

- it appears in a named common block in a block data program unit.

- if it appears in a named common block in a module.

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 **initialization_expr** or **NULL()** is specified, and the entity you are declaring:

- is a variable, the variable is initially defined.

**Fortran 95**

- is a derived type component, the derived type has default initialization.

**End of Fortran 95**

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 **ALLOCATABLE** or **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.

**IBM Extension**

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.

**End of IBM Extension**

**Examples**

`COMPLEX, DIMENSION (2,3) :: ABC(3) ! ABC has 3 (not 6) array elements`

**Related information**

- “Complex” on page 22
- “Initialization expressions” on page 85
- “How type is determined” on page 51, for details on the implicit typing rules
- “Array declarators” on page 65
- “Automatic objects” on page 16
- “Storage classes for variables” on page 59
- “DATA” on page 273, 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.

**Syntax**

```
CONTAINS
```

**Rules**

For a CONTAINS statement associated with subprograms, the following rules apply:

- 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
MODULE A
    ...
    CONTAINS
    SUBROUTINE B(X)
    ...
    CONTAINS
    FUNCTION C(Y)
    ...
END FUNCTION
END SUBROUTINE
END MODULE
```

**Related information**

- “Program units, procedures, and subprograms” on page 138

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.

**Syntax**
Examples

DO 100 I = 1, N
   X = X + N
100 CONTINUE

Related information

- Chapter 6, “Execution Control,” on page 117

CYCLE

Purpose

The CYCLE statement terminates the current execution cycle of a DO or DO WHILE construct.

Syntax

```
CYCLE [DO_construct_name]
```

- `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
```
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 280
- “DO WHILE” on page 282

DATA

Purpose

The DATA statement provides initial values for variables.

Syntax

```
DATA data_object_list / initial_value_list /
```

*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_list, do_variable = integer_expr1, integer_expr2)
```

*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. If $r$ is a named constant, it must have been declared previously in the scoping unit or made accessible by use or host association.

**Fortran 95**

$r$ is also a nonnegative scalar integer subobject of a constant. Similar to the above paragraph, if it is a subobject of a named constant, it must have been declared previously in the scoping unit or made accessible by use or host association.

**End of Fortran 95**

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, signed integer literal constant, signed real literal constant, structure constructor, scalar subobject of a constant, or NULL().

**Rules**

Specifying a nonpointer 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.

**Fortran 95**

An array with pointer attribute has only one corresponding initial value which is NULL().

**End of Fortran 95**

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 $data\_object$.

**Fortran 95**

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().

**End of Fortran 95**
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 a subobject of a named constant, the named constant must have been previously declared in the scoping unit, or made accessible by host or use association. If data_value is a structure constructor, the derived type must have been previously declared in the scoping unit, or made accessible by host or use 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.

IBM Extension

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 it as the named constant and assigns its value to the corresponding list item in the DATA statement.

End of IBM Extension

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:

```fortran
INTEGER Z(100),EVEN_ODD(0:9)
LOGICAL FIRST_TIME
CHARACTER*10 CHARARR(1)
DATA FIRST_TIME / .TRUE. /
DATA Z / I00* 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:

```fortran
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 15
- “Executing a DO statement” on page 120
- “Statement and construct entities” on page 132

DEALLOCATE

Purpose

The DEALLOCATE statement dynamically deallocates allocatable objects and pointer targets. A specified pointer becomes disassociated, while any other pointers associated with the target become undefined.

Syntax
object is a data pointer or an allocatable object

stat_variable is a scalar integer variable

Rules

An allocatable object that appears in a DEALLOCATE statement must be currently allocated. An allocatable object with the TARGET attribute cannot be deallocated through an associated pointer. Deallocation of such an object causes the association status of any associated pointer to become undefined. An allocatable object 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 object to become not allocated.

Tips

Use the DEALLOCATE statement instead of the NULLIFY statement if no other pointer is associated with the allocated memory.

Deallocation 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:

<table>
<thead>
<tr>
<th>Stat value</th>
<th>Error condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No error</td>
</tr>
<tr>
<td>1</td>
<td>Error in system routine attempting to do deallocation</td>
</tr>
<tr>
<td>2</td>
<td>An invalid data object has been specified for deallocation</td>
</tr>
<tr>
<td>3</td>
<td>Both error conditions 1 and 2 have occurred</td>
</tr>
</tbody>
</table>

---

IBM Extension

End of IBM Extension
An allocate_object must not depend on the value, bounds, allocation status, or association status of another allocate_object in the same DEALLOCATE statement; nor does it depend on the value of the stat_variable in the same DEALLOCATE statement.

stat_variable must not be deallocated within the same DEALLOCATE statement. The variable must not depend on the value, bounds, allocation status, or association status of any allocate_object in the same DEALLOCATE statement.

Examples

```
INTEGER, ALLOCATABLE :: A(:, :)
INTEGER X, Y
  
  ALLOCATE (A(X, Y))
  
  DEALLOCATE (A, STAT=1)
END
```

Related information

- "ALLOCATE" on page 240
- "ALLOCATABLE" on page 238
- "Allocation status" on page 58
- "Pointer association" on page 136
- "Deferred-shape arrays" on page 68
- "Allocatable objects as dummy arguments" on page 166
- "Allocatable components" on page 36

Derived Type

Purpose

The Derived Type statement is the first statement of a derived-type definition.

Syntax

```
TYPE type_attribute_list :: type_name
```

*type_attribute* is PRIVATE, PUBLIC, BIND(C), .

*type_name* is the name of the derived type

Rules

PRIVATE or PUBLIC can only be specified if the derived-type definition is within the specification part of a module. Only one of PRIVATE or PUBLIC may be specified.
BIND(C) explicitly defines the Fortran derived type as interoperable with a C type. The components must be of interoperable types. (See “Interoperability of types” on page 679 for additional information.) A derived type with the BIND attribute cannot be a SEQUENCE type. A component of a derived type with the BIND attribute must have interoperable type and type parameters, and cannot have the POINTER or ALLOCATABLE attribute.

The type_name must not be the same as the name of any intrinsic type, except BYTE and DOUBLECOMPLEX. The type_name must also not be 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 28
- “Interoperability of types” on page 679
- “END TYPE” on page 297
- “SEQUENCE” on page 396
- “PRIVATE” on page 369

DIMENSION

Purpose

The DIMENSION attribute specifies the name and dimensions of an array.

Syntax

```
DIMENSION array_declarator_list
```

Rules

According to Fortran 95, you can specify an array with up to seven dimensions.
IBM Extension

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
- BIND
- INTENT
- OPTIONAL
- PARAMETER
- POINTER
- PRIVATE
- PROTECTED
- PUBLIC
- SAVE
- STATIC
- TARGET
- VOLATILE

Examples

CALL SUB(5,6)
CONTAINS
SUBROUTINE SUB(I,M)
   DIMENSION LIST1(I,M)
   INTEGER, ALLOCATABLE, DIMENSION(:, :) :: A
   ! automatic array
   INTEGER, DIMENSION(:, :) :: M
   ! deferred-shape array
   
   : :
END SUBROUTINE
END

Related information

- Chapter 4, “Array concepts,” on page 63
- "VIRTUAL" on page 419

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.

Syntax

\[
\text{DO}_\text{construct\_name}:: \text{stmt\_label} \quad \text{var\_name} = \text{a\_expr1, a\_expr2,} \ldots, \text{a\_exprn}
\]

\text{DO}_\text{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).

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 119
- “END (Construct)” on page 293, for details on the END DO statement
- “EXIT” on page 307
- “CYCLE” on page 272
- “INDEPENDENT” on page 444
- “ASSERT” on page 436
- “CNCALL” on page 440
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.

Syntax

```
DO_construct_name: stmt_label, WHILE (logical_expr)
```

- **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 123
- “END (Construct)” on page 293 for details on the END DO statement
- “EXIT” on page 307
- “CYCLE” on page 272

DOUBLE COMPLEX

IBM Extension

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.

Syntax

```
DOUBLE COMPLEX
  entity_decl_list
    attr_spec_list

where:

attr_spec

ALLOCATABLE
AUTOMATIC
BIND
DIMENSION (array_spec)
EXTERNAL
INTENT (intent_spec)
INTRINSIC
OPTIONAL
PARAMETER
POINTER
PRIVATE
PUBLIC
SAVE
STATIC
TARGET
VOLATILE
```
For detailed information on rules about a particular attribute, refer to the statement of the same name.

is either IN, OUT, or INOUT

: is the double colon separator. Use the double colon separator when you specify attributes, =initialization_expr or => NULL

is a list of dimension bounds

e is an object name or function name. array_spec cannot be specified for a function with an implicit interface.

provides an initial value for the entity specified by the immediately preceding name

provides an initial value, by means of an initialization expression, for the entity specified by the immediately preceding name

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 object, 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 ALLOCATABLE or 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**

```fortran
SUBROUTINE SUB
  DOUBLE COMPLEX, STATIC, DIMENSION(1) :: B
END SUBROUTINE
```

**Related information**

- “COMPLEX” on page 266
- “Initialization expressions” on page 85
- “How type is determined” on page 51, for details on the implicit typing rules
- “Array declarators” on page 65
- “Automatic objects” on page 16
- “Storage classes for variables” on page 59
- “DATA” on page 273, 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.

Syntax

```
DOUBLE PRECISION entity_decl_list
          attr_spec_list
```

where:

```
attr_spec
ALLOCATABLE
AUTOMATIC
BIND
DIMENSION (array_spec)
EXTERNAL
INTENT (intent_spec)
INTRINSIC
OPTIONAL
PARAMETER
POINTER
PRIVATE
PUBLIC
SAVE
STATIC
TARGET
VOLATILE
```

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. Use the double colon separator when you
specify attributes, =initialization_expr, or => NULL().
```

```
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 with an implicit interface.

**IBM Extension**

initial_value provides an initial value for the entity specified by the immediately preceding name

**End of IBM Extension**

initialization_expr provides an initial value, by means of an initialization expression, for the entity specified by the immediately preceding name

**Fortran 95**

=> NULL() provides the initial value for the pointer object

**End of Fortran 95**

**Rules**

**Fortran 95**

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 object, 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.

Fortran 95

In Fortran 95, a pointer can be initialized. Pointers can only be initialized by the use of => NULL().

End of Fortran 95

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.

Fortran 95

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 NULL0 is specified, the variable is initially defined. If the entity you are declaring is a derived type component, and initialization_expr or NULL0 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 => NULL0 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.

End of Fortran 95

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 385
- “Initialization expressions” on page 85
- “How type is determined” on page 51
- “Array declarators” on page 65
- “Automatic objects” on page 16
- “Storage classes for variables” on page 59
- “DATA” on page 273

ELSE

Purpose

The ELSE statement is the first statement of the optional ELSE block within an IF construct.

Syntax

```
ELSE
  IF_construct_name
```

`IF_construct_name`

is a name that identifies the IF construct

Syntax

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 124
- “END (Construct)” on page 293, 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.

**Syntax**

```
ELSE IF (scalar_logical_expr) THEN
  IF_construct_name
```

*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
```

### ELSEWHERE

**Purpose**

The ELSEWHERE statement is the first statement of the optional ELSEWHERE or masked ELSEWHERE block within a WHERE construct.

**Related information**

- “IF construct” on page 124
- “END (Construct)” on page 293, for details on the END IF statement
- “ELSE” on page 289
Syntax

```
ELSEWHERE
  (∼mask_expr∼)
where_construct_name
```

Notes:
1  Fortran 95.
2  Fortran 95.

```
Fortran 95

mask_expr is a logical array expression
```

```
Fortran 95

where_construct_name is a name that identifies a WHERE construct
```

Rules

```
Fortran 95

A masked ELSEWHERE statement contains a mask_expr. See “Interpreting masked array assignments” on page 105 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:

```fortran
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 |
```

Chapter 10. Statements and attributes 291
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:
!
<table>
<thead>
<tr>
<th>ARR1</th>
<th>ARR2</th>
<th>FLAG</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 4 7</td>
<td>9 6 3</td>
<td>-1 -1 1</td>
</tr>
<tr>
<td>2 5 8</td>
<td>8 5 2</td>
<td>-1 0 1</td>
</tr>
<tr>
<td>3 6 9</td>
<td>7 4 1</td>
<td>-1 1 1</td>
</tr>
</tbody>
</table>

Related information

- “WHERE construct” on page 103
- “WHERE” on page 423
- “END (Construct)” on page 293, for details on the END WHERE statement

END

Purpose

An END statement indicates the end of a program unit or procedure.

Syntax

```fortran
>>>END

-BLOCK DATA_ BLOCK_DATA_name
-FUNCTION FUNCTION_name
-MODULE MODULE_name
-PROGRAM PROGRAM_name
-SUBROUTINE SUBROUTINE_name
```

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

```fortran
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 129

END (Construct)

Purpose

The END (Construct) statement terminates the execution of a construct. The Construct Termination Statements table lists the appropriate statement to end each construct.

Table 30. Construct termination statements

<table>
<thead>
<tr>
<th>Construct</th>
<th>Termination Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSOCIATE</td>
<td>END ASSOCIATE</td>
</tr>
<tr>
<td>DO</td>
<td>END DO</td>
</tr>
<tr>
<td>DO WHILE</td>
<td>END DO</td>
</tr>
<tr>
<td>FORALL</td>
<td>END FORALL</td>
</tr>
<tr>
<td>IF</td>
<td>END IF</td>
</tr>
<tr>
<td>SELECT CASE</td>
<td>END SELECT</td>
</tr>
<tr>
<td>WHERE</td>
<td>END WHERE</td>
</tr>
</tbody>
</table>
The `END FORALL` statement terminates `FORALL` constructs.

### Syntax

```
END ASSOCIATE
  ASSOCIATE_construct_name
END DO
  DO_construct_name
END FORALL
  FORALL_construct_name
END IF
  IF_construct_name
END SELECT
  CASE_construct_name
    SELECT_TYPE_construct_name
    where_construct_name
END WHERE
```

### Notes:

1. Fortran 2003 Standard
2. Fortran 95.
3. Fortran 95.

### Fortran 2003 Standard

- `ASSOCIATE_construct_name` is a name that identifies an `ASSOCIATE` construct
- `DO_construct_name` is a name that identifies a `DO` or `DO WHILE` construct

### Fortran 95

- `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 SELECT 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 ASSOCIATE, END DO, END IF, or END SELECT statement from within the ASSOCIATE, 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
  I=I-1
END DO DECR
  ! Cannot reference a construct name
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, “Execution Control,” on page 117
- “ASSOCIATE Construct” on page 117
- “DO” on page 280
- “FORALL” on page 311
- “FORALL (construct)” on page 314
- “IF (block)” on page 325
- “SELECT CASE” on page 395
- “WHERE” on page 423
- “Deleted features” on page 746

END INTERFACE

Purpose

The END INTERFACE statement terminates a procedure interface block.

Syntax

```
END INTERFACE
```

Notes:
1. Fortran 95.

```
generic_spec
```

```
generic_name
    OPERATOR ( defined_operator )
    ASSIGNMENT ( = )
End of Fortran 95
```

```
defined_operator
```

is a defined unary operator, defined binary operator, or extended intrinsic operator.

```
End of Fortran 95
```

END OF INTERFACE
Rules

Each INTERFACE statement must have a corresponding END INTERFACE statement.

An END INTERFACE statement without a generic_spec can match any INTERFACE statement, with or without a generic_spec.

---

**Fortran 95**

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

```fortran
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) Y
    REAL Y(50,50), INVERSE
  END FUNCTION
END INTERFACE OPERATOR (.INVERSE.)
```

---

Related information

- "INTERFACE” on page 343
- "Interface concepts” on page 139

---

**END TYPE**

**Purpose**

The END TYPE statement indicates the completion of a derived-type definition.
Syntax

```fortran
END TYPE
```

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

```fortran
TYPE A
  INTEGER :: B
  REAL :: C
END TYPE A
```

Related information

- “Derived types” on page 28

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.

An `ENDFILE` statement for a file connected for stream access causes the terminal point to become the current file position. File storage units before the current position are considered written, and can be read. You can write additional data to the file by using subsequent stream output statements.

Syntax

```fortran
ENDFILE (position_list)
```

- `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*.

### Fortran 2003 Standard

**IOMSG=** *iomsg_variable*

is an input/output status specifier that specifies the message returned by the input/output operation. *iomsg_variable* is a scalar default character variable. It must not be a use-associated nonpointer protected variable. When the input/output statement containing this specifier finishes execution, *iomsg_variable* is defined as follows:

- If an error, end-of-file, or end-of-record condition occurs, the variable is assigned an explanatory message as if by assignment.
- If no such condition occurs, the value of the variable is unchanged.

### End of Fortran 2003 Standard

**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

#### IBM Extension

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.

#### End of IBM Extension

After execution of an **ENDFILE** statement for a file connected for sequential access, a **BACKSPACE** or **REWIND** statement must be used to reposition the file prior to execution of any data transfer input/output statement.

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*.

#### IBM Extension

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

```fortran
ENDFILE 12
ENDFILE (IOSTAT=IOSS,UNIT=11)
```

Related information

- “Conditions and IOSTAT values” on page 186
- Chapter 8, “XL Fortran Input/Output,” on page 177
- Setting Run-time Options in the XL Fortran Compiler Reference

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.

Syntax

```
ENTRY entry_name([dummy_argument_list])
RESULT(result_name)
BIND(C, NAME = binding_label)
```

- `entry_name` is the name of an entry point in a function subprogram or subroutine subprogram
- `binding_label` is a scalar character initialization expression
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.

IBM Extension

The ENTRY statement cannot appear in a CRITICAL, MASTER, PARALLEL, PARALLEL SECTIONS, SECTIONS, or SINGLE construct.

End of IBM Extension

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, nonallocatable 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 PARAMETER, INTENT, OPTIONAL, SAVE, or VOLATILE attributes. The STATIC and AUTOMATIC attributes can be specified only when the result variable is not an allocatable object, 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 must not be initialized in a type declaration statement or DATA 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.

At most one RESULT clause and at most one BIND clause may appear. They can appear in any order.

Fortran 2003 Standard

The BIND keyword implicitly or explicitly defines a binding label which specifies the name by which an entity is accessed from the C programming language. The result variable, if there is a result, must be a scalar that is interoperable. A binding label cannot be specified for a dummy argument. A dummy argument cannot be
zero-sized. A dummy argument for a procedure with the BIND attribute must have interoperable types and type parameters, and cannot have the ALLOCATABLE, OPTIONAL, or POINTER attribute.

End of Fortran 2003 Standard

In a function subprogram, entry_name identifies a function and 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.

End of Fortran 95

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.

End of Fortran 95

If entry_name is of type character, its length cannot be an asterisk if the function is recursive.

IBM Extension

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**

```fortran
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 317
- **SUBROUTINE** on page 401
- **Recursion** on page 171
- **Dummy arguments** on page 159
- `-qrecur Option` in the XL Fortran Compiler Reference

---

**ENUM/END ENUM**

**Purpose**

You can specify an **ENUM** statement to define and group a set of named integer constants. The named integer constants in an **ENUM** statement are called enumerators.

**Syntax**

To define an enumerator, you must use an enumeration construct:

```
Enumeration construct
   `--ENUM, BIND(C)`
   `--enumeration_block`
   `--ENDENUM`
```
If you want to specify an enumerator with a *scalar_initialization_expression*, you must also specify a `::`.

**Rules**

If you specify a scalar integer initialization expression, the value of the enumerator is the result of the scalar integer initialization expression.

---

**IBM Extension**

You can only use a logical operator if you compile with `-qintlog`.

---

If you do not specify a scalar integer initialization expression and the enumerator is first in the definition body of the type, the value of the enumerator is 0.

If you do not specify a scalar integer initialization expression and the enumerator is after another enumerator in the definition body of the type, the value is one greater than the value of the preceding enumerator.

You can set the kind type parameter of an enumerator using the `-qenum` option. If you do not specify `-qenum`, the default kind for an enumerator is 4.

**Examples**

The following example uses the **ENUM** statement in different ways to define enumerators.

```
enum, bind(c)
  enumerator :: red = 1, blue, black = 5
  enumerator yellow
  enumerator gold, silver, bronze
  enumerator :: purple
  enumerator :: pink, lavender
end enum
```

The values of these enumerators are: red = 1, blue = 2, black = 5, yellow = 6, gold = 7, silver = 8, bronze = 9, purple = 10, pink = 11, lavender = 12.

If you supply an initial value for an enumerator, then a `::` is required in the **ENUMERATOR** statement. The `red` and `black` enumerators in the list are initialized with a scalar integer initialization expression.

The `::` is optional in an enumerator definition when scalar integer initialization expressions are not used to initialize any of the enumerators in the list of enumerators being declared.
• In the second and third enumerator definitions, the :: is not necessary as yellow, gold, silver, and bronze are not initialized with a scalar integer initialization expression.
• The fourth and fifth enumerator definitions show that :: can be used even when purple is not initialized with a scalar integer initialization expression.

Related information
• “PARAMETER” on page 362

---

EQUIVALENCE

Purpose
The EQUIVALENCE statement specifies that two or more objects in a scoping unit are to share the same storage.

Syntax

```
EQUIVALENCE (equiv_object, equiv_object_list)
```

`equiv_object` is a variable name, array element, or substring. Any subscript or substring expression must be an integer initialization expression. A substring cannot have a length of zero.

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 object, object of nonsequence derived type, object of sequence derived type that contains a pointer or allocatable component, or a subobject of any of these.

---

Variables with the [BIND] attribute, or variables that are members of a common block with the BIND attribute must not be an object in an EQUIVALENCE statement.

---

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.

<table>
<thead>
<tr>
<th>IBM Extension</th>
</tr>
</thead>
</table>

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.

<table>
<thead>
<tr>
<th>End of IBM Extension</th>
</tr>
</thead>
</table>

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.

| F2003 | If you specify that an object declared by an **EQUIVALENCE** group has the PROTECTED attribute, all objects specified in that **EQUIVALENCE** group must have the PROTECTED attribute. | F2003 |

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

```fortran
DOUBLE PRECISION A(3)
REAL B(5)
EQUIVALENCE (A,B(3))
```

Association of storage units:

```
<table>
<thead>
<tr>
<th>A(1)</th>
<th>A(2)</th>
<th>A(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(1)</td>
<td>B(2)</td>
<td>B(3)</td>
</tr>
<tr>
<td>B(4)</td>
<td>B(5)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A(1)</td>
<td>A(2)</td>
</tr>
<tr>
<td></td>
<td>B(1)</td>
<td>B(2)</td>
</tr>
</tbody>
</table>
```

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:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable A:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable B:</td>
<td>C(1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Array C:</td>
<td></td>
<td>C(2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 52

EXIT

Purpose

The EXIT statement terminates execution of a DO construct or DO WHILE construct before the construct completes all of its iterations.

Syntax

```
EXIT
```

where `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**

```fortran
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 LOOP3
```

**Related information**

- [“DO construct” on page 119](#)
- [“DO WHILE construct” on page 123](#)

---

**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.

**Syntax**

```fortran
CALL EXTERNAL name_list
```

- `name_list` 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

```fortran
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 167
- Item 4 under Appendix A, “Compatibility across standards,” on page 743

---

**FLUSH**

**Fortran 2003 Standard**

### Purpose

The **FLUSH** statement makes data written to an external file available to other processes, or causes data placed in an external file by means other than Fortran to be available to a **READ** statement.

### Syntax

```
&FLUSH& u

flush_list
```

- **u** is an integer scalar expression with a value in a range from 0 through 2,147,483,647. This unit references an external file. The value must not be an asterisk or a **Hollerith constant**.

- **flush_list** is a list of specifiers that must contain **UNIT=**, and can also contain one of each of the following specifiers:
  - **[UNIT=]** specifies the external file as an integer scalar expression with a value in a range from 0 through 2,147,483,647. The value of must not be an asterisk or a Hollerith constant.
  - **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. Inclusion of the **ERR=** specifier suppresses error messages. **stmt_label** must be the statement label of a
branch target statement that appears in the same scoping unit as the FLUSH statement.

---

**Fortran 2003 Standard**

- IOMSG=iomsg_variable is an input/output status specifier that specifies the message returned by the input/output operation. iomsg_variable is a scalar default character variable. It must not be a use-associated nonpointer protected variable. When the input/output statement containing this specifier finishes execution, iomsg_variable is defined as follows:
  - If an error, end-of-file, or end-of-record condition occurs, the variable is assigned an explanatory message as if by assignment.
  - If no such condition occurs, the value of the variable is unchanged.

---

**End of Fortran 2003 Standard**

- IOSTAT=ios specifies the status of the flush operation as a scalar variable of type INTEGER. When execution of the flush statement completes, ios is:
  - A zero value if no error condition occurs.
  - A positive value if an error occurs.
  - A negative value if the device cannot perform a flush operation, such as a tape or TTY device.

Inclusion of the IOSTAT specifier suppresses error messages. If the program encounters a severe error, the value of ios is 200.

If you do not specify ERR or IOSTAT, the program terminates on encountering a severe error.

**Rules**

The FLUSH statement must not appear in a pure subprogram.

A FLUSH statement has no effect on file position.

The buffering run-time option does not affect the execution of the FLUSH statement.

**Examples**

**Example 1:** In the following example a data file written by a Fortran program is read by a C routine. The program specifies a FLUSH statement for the buffered I/O.

```fortran
! The following Fortran program writes data to an external file.
subroutine process_data()
    integer data(10)
    external read_data

    data = ((i,i=1,10))
    open(50, file="data_file")
    write(50, *) data
    flush(50) ! since Fortran I/O is buffered, a FLUSH statement is needed for the C routine to read the data
    call read_data(10)
end subroutine
```

`! call C routine to read the file`
/* The following C routine reads data from the external file. */
void read_data(int *sz) {

#include <stdio.h>
#include <stdlib.h>
int *data, i;
FILE *fp;

data = (int *) malloc(*sz*sizeof(int));
fp = fopen("data_file", "r");
for (i=0; i<*sz-1; i++) {
    fscanf(fp, "%d", &data[i]);
}
}

End of Fortran 2003 Standard

FORALL

Fortran 95

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.

Syntax

```
FORALL(forall_header)forall_assignment
```

```
forall_header
```

```
(forall_triplet_spec_list), scalar_mask_expr
```

```
forall_triplet_spec
```

```
index_name = subscript : subscript : stride
```

```
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:
   \text{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 -l 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)
```

**IBM Extension**

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.

```
END OF IBM Extension
```

**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 100
- “Pointer assignment” on page 112
- “FORALL construct” on page 109
- “INDEPENDENT” on page 444
- “Statement and construct entities” on page 132

-- End of Fortran 95

FORALL (construct)

Purpose

The FORALL (Construct) statement is the first statement of the FORALL construct.

Syntax

```
FORALL_construct_name : FORALL 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 \( max \leq 0 \) for any index, *forall_assignment* is not executed. For example:

```
index1 = 2:10:3   ! The index values are 2,5,8.
      = 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 293
- “FORALL construct” on page 109
- “Statement and construct entities” on page 132

---

**FORMAT**

**Purpose**

The FORMAT statement provides format specifications for input/output statements.

**Syntax**

```
FORMAT( format_item_list )
```

`format_item`
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

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 379) 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:

IBM Extension

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.

End of IBM Extension

Examples

```
CHARACTER*32 CHARVAR
CHARVAR="('integer: ',I2,' binary: ',8B)" ! Character format
M = 56 ! specification
J = 1
X = 2355.95843
WRITE (6,770) M,X
WRITE (6,CHARVAR) M,M
WRITE (6,880) J,M

770 FORMAT(I3, 2F10.2)
880 FORMAT(I<J+1>)
END
```

Related information

- Chapter 9, “Input/Output formatting,” on page 195
- “PRINT” on page 367
- “READ” on page 377
- “WRITE” on page 425

FUNCTION

Purpose

The FUNCTION statement is the first statement of a function subprogram.

Syntax
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 409 for details about **type_spec**.

**name** is the name of the function subprogram.

**len** is either an unsigned integer literal or a parenthesized scalar integer initialization expression. Its value specifies the length of the function’s result variable. It can be included only when the type is specified in the **FUNCTION** statement. The type cannot be **DOUBLE PRECISION**, **DOUBLE COMPLEX**, **BYTE**, or a derived type.

---

**Notes:**
1 IBM Extension.
2 IBM Extension.

---

**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 409 for details about **type_spec**.

**name** is the name of the function subprogram.

**len** is either an unsigned integer literal or a parenthesized scalar integer initialization expression. Its value specifies the length of the function’s result variable. It can be included only when the type is specified in the **FUNCTION** statement. 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.

At most one **RESULT** clause and at most one **BIND** clause may appear. They can appear in any order.

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.

### Fortran 2003 Standard

The **BIND** keyword implicitly or explicitly defines a binding label by which a procedure is accessed from the C programming language. The result variable must be a scalar that is interoperable. A dummy argument cannot be zero-sized. A dummy argument for a procedure with the **BIND** attribute must have interoperable types and type parameters, and cannot have the **ALLOCATABLE**, **OPTIONAL**, or **POINTER** attribute.

The **BIND** attribute must not be specified for an internal procedure. If the **FUNCTION** statement appears as part of an interface body that describes a dummy procedure, the `NAME=` specifier must not appear. An elemental procedure cannot have the **BIND** attribute.

### End of Fortran 2003 Standard

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 **PARAMETER**, **INTENT**, **OPTIONAL**, or **SAVE** attributes. The **STATIC** and **AUTOMATIC** attributes can be specified only when the result variable is not an allocatable object, an array or a pointer, and 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 **ALLOCATABLE** or **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.

--- IBM Extension ---

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.

--- End of IBM Extension ---

Elemental procedures

--- Fortran 95 ---

For elemental procedures, the keyword ELEMENTAL must be specified. If the ELEMENTAL keyword is specified, the RECURSIVE keyword cannot be specified.

--- End of Fortran 95 ---

Examples

```fortran
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 154
- “ENTRY” on page 300
- “BIND” on page 247
- “Function reference” on page 155
- “Dummy arguments” on page 159
- “Statement Function” on page 397
- “Recursion” on page 171
- -qrecur option in the XL Fortran Compiler Reference
- “Pure procedures” on page 171
- “Elemental Procedures” on page 174

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.

Syntax

```fortran
GO TO variable_name (stmt_label_list)
```

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.

---

**Fortran 95**

The assigned GO TO statement has been deleted in Fortran 95.

---

**Examples**

```fortran
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**

- "Statement labels" on page 5
- "Branching" on page 127
- "Deleted features" on page 746

---

**GO TO (computed)**

**Purpose**

The computed GO TO statement transfers program control to one of possibly several executable statements.

**Syntax**

```
GO TO (stmt_label_list), arith_expr
```

*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 expression.
IBM Extension

It can also be real or complex. If the value of the expression is noninteger, XL Fortran converts it to INTEGER(4) before using it.

End of IBM Extension

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 5
- “Branching” on page 127

GO TO (unconditional)

Purpose

The unconditional GO TO statement transfers program control to a specified executable statement.

Syntax

```plaintext
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 5
- “Branching” on page 127

IF (arithmetic)

Purpose

The arithmetic IF statement transfers program control to one of three executable statements, depending on the evaluation of an arithmetic expression.

Syntax

```
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
```
IF (block)

Purpose
The block IF statement is the first statement in an IF construct.

Syntax

```
IF_construct_name : IF(scalar_logical_expr) THEN
```

* **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
```
WHICH:C: 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 124
- “ELSE IF” on page 290
- “ELSE” on page 289
- “END (Construct)” on page 293, 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.

Syntax

```
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, ASSOCIATE construct statement, FORALL construct statement, or WHERE construct statement.

Examples

```
IF (ERR.NE.0) CALL ERROR(ERR)
```

Related information

Chapter 6, “Execution Control,” on page 117

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.

Syntax
**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.

---
**IBM Extension**

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.

---
**End of IBM Extension**
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.

**IBM Extension**

**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.

**End of IBM Extension**

An **IMPLICIT** statement does not change the data type of an intrinsic function.

**IBM Extension**

Using the `-qsave/-qnosave` compiler option modifies the predefined conventions for storage class:

<table>
<thead>
<tr>
<th>Compiler Option</th>
<th>Predefined Convention</th>
<th><strong>IMPLICIT</strong> Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-qsave</code></td>
<td>makes the predefined convention</td>
<td>STATIC( a - _)</td>
</tr>
<tr>
<td><code>-qnosave</code></td>
<td>makes the predefined convention</td>
<td>AUTOMATIC( a - _)</td>
</tr>
</tbody>
</table>

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.

**End of IBM Extension**

**Examples**

```fortran
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
```

---

328 XL Fortran Language Reference
Related information

- "How type is determined" on page 51 for a discussion of the implicit rules
- "Storage classes for variables" on page 59
- -qundef option in the XL Fortran Compiler Reference
- -qsave option in the XL Fortran Compiler Reference

IMPORT

Fortran 2003 Standard

Purpose

The IMPORT statement makes named entities from the host scoping unit accessible in the interface body by host association.

Syntax

```
IMPORT import_name_list
```

`import_name_list` is a list of named entities that are accessible in the host scoping unit.

Rules

The IMPORT statement is allowed only in an interface body. Each of the specified names must be explicitly declared before the interface body.

The entities in the import name list are imported into the current scoping unit and are accessible by host association. If no names are specified, all of the accessible named entities in the host scoping unit are imported.

The names of imported entities must not appear in any context that would cause the host entity to be inaccessible.

Examples

```
use, intrinsic :: ISO_C_BINDING
interface
  subroutine process_buffer(buffer, n_bytes), bind(C,NAME="ProcessBuffer")
    IMPORT :: C_PTR, C_INT
    type (C_PTR), value :: buffer
    integer (C_INT), value :: n_bytes
  end subroutine process_buffer
end interface
```

End of Fortran 2003 Standard
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.
- Inquire by output list, which requires the IOLENGTH= specifier
- Inquire by unit, which requires the UNIT= specifier.

Syntax

\[
\text{INQUIRE} (\text{inquiry_list}) (\text{IOLENGTH} = \text{iol}) \text{output_item_list}
\]

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:

\[\text{[UNIT=} \text{u}\]

is a unit specifier. It specifies the unit about which the inquire-by-unit form of the statement is inquiring. \text{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 an integer expression, whose value is in the range 0 through 2147483647. If the optional characters UNIT= are omitted, \text{u} must be the first item in inquiry_list.

\[\text{ACCESS=} \text{char_var}\]

indicates whether the file is connected for direct access, sequential access, or stream access. \text{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. The value assigned is STREAM if the file is connected for stream access. If there is no connection, \text{char_var} is assigned the value UNDEFINED.

\[\text{ACTION=} \text{act}\]

indicates if the file is connected for read and/or write access. \text{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.

---

**IBM Extension**

**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.

---

**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**.

**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.

**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.

**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.

**EXIST** = *ex*

indicates if a file or unit exists. *ex* is an integer variable 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.

**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 Linux operating system file name. The named file does not have to exist, nor does it have to be associated with a unit.
IBM Extension

**Note:** A valid Linux 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).

End of IBM Extension

**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.

**FORMATED= fnt**
indicates if the file can be connected for formatted input/output. fnt 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.

Fortran 2003 Standard

**IOMSG= iomsg_variable**
is an input/output status specifier that specifies the message returned by the input/output operation. iomsg_variable is a scalar default character variable. It must not be a use-associated nonpointer protected variable. When the input/output statement containing this specifier finishes execution, iomsg_variable is defined as follows:

- If an error, end-of-file, or end-of-record condition occurs, the variable is assigned an explanatory message as if by assignment.
- If no such condition occurs, the value of the variable is unchanged.

End of Fortran 2003 Standard

**IOSTAT= ios**
is an input/output status specifier that specifies the status of the input/output operation. ios is an integer variable. 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.

**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.

**NAMED= nmd**
indicates if the file has a name. nmd is an integer variable 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.

**NEXTREC= nr**
indicates where the next record can be read or written on a file connected for direct access. nr is an integer variable 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.

---

**IBM Extension**

Because record numbers can be greater than \( 2^{31} - 1 \), you may choose to make the scalar variable specified with the `NXTREC=` 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.

---

**NUMBER= num**

indicates the external unit identifier currently associated with the file. \( num \) is an integer variable 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.

**OPENED= od**

indicates if a file or unit is connected. \( od \) is an integer variable 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= character_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= character_var` 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.

**PAD= pd**

indicates if the connection of the file had specified `PAD=YES`. \( pd \) 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.

**POS=integer_var**

\[ \text{F2003} \]

`integer_var` is an integer variable that indicates the value of the file position for a file connected for stream access. `integer_var` is assigned the number of the file storage unit immediately following the current position of a file connected for stream access. If the file is positioned at its terminal position, `integer_var` is assigned a value one greater than the highest-numbered storage unit in the file. `integer_var` becomes undefined if the file is not connected for stream access or if the position of the file cannot be determined because of previous error conditions. \[ \text{F2003} \]

**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 **REWOOD**. 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**.

**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.

**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.

**RECL** = *rci*

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.

*rci* is an integer variable 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, *rci* becomes undefined.

If the file is connected for stream access, *rci* becomes undefined.

**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.

**SIZE** = *filesize*

*filesize* is an integer variable that is assigned the file size in bytes.

---

**Fortran 2003 Standard**

---

**STREAM** = *strm*

is a scalar default character variable that indicates whether the file is connected for stream access. *strm* is assigned the value **YES** if the file can be accessed using stream access, the value **NO** if the file cannot be accessed using stream access, or the value **UNKNOWN** if access cannot be determined.

---

**End of Fortran 2003 Standard**

---

**IBM Extension**
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.

c_char_variable is a scalar character variable. If char_variable is assigned the value BOTH, then both synchronous and synchronous 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.

End of IBM Extension

UNFORMATTED= unf
indicates if the file can be connected for unformatted input/output. fnt 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.

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.

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.

IBM Extension

If the unit or file is connected, the values returned for the ACCESS=, SEQUENTIAL=, STREAM=, 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=, STREAM=, 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. The value returned by the SIZE= specifier is -1.

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.

End of IBM Extension

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 186
- Chapter 8, “XL Fortran Input/Output,” on page 177

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.

Syntax

```
 INTEGER [kind_selector] entity_decl_list
```

```
, attr_spec_list =::
```

336  XL Fortran Language Reference
where:

<table>
<thead>
<tr>
<th>attr_spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALLOCATABLE</td>
</tr>
<tr>
<td>AUTOMATIC</td>
</tr>
<tr>
<td>BIND</td>
</tr>
<tr>
<td>DIMENSION (array_spec)</td>
</tr>
<tr>
<td>EXTERNAL</td>
</tr>
<tr>
<td>INTENT (intent_spec)</td>
</tr>
<tr>
<td>INTRINSIC</td>
</tr>
<tr>
<td>OPTIONAL</td>
</tr>
<tr>
<td>PARAMETER</td>
</tr>
<tr>
<td>POINTER</td>
</tr>
<tr>
<td>PRIVATE</td>
</tr>
<tr>
<td>PUBLIC</td>
</tr>
<tr>
<td>SAVE</td>
</tr>
<tr>
<td>STATIC</td>
</tr>
<tr>
<td>TARGET</td>
</tr>
<tr>
<td>VOLATILE</td>
</tr>
</tbody>
</table>

kind_selector

```
( KIND = int_initialization_expr )
* int_literal_constant
```

Notes:
1. IBM Extension.

IBM Extension

specifies the length of integer entities: 1, 2, 4 or 8. *int_literal_constant* cannot specify a kind type parameter.

End of IBM Extension

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. Use the double colon separator when you specify attributes, =initialization_expr or => NULL().

array_spec

is a list of dimension bounds

entityDecl
a is an object name or function name. array_spec cannot be specified for a function name with an implicit interface.

--- IBM Extension ---

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.

--- End of IBM Extension ---

--- IBM Extension ---

initial_value provides an initial value for the entity specified by the immediately preceding name.

--- End of IBM Extension ---

initialization_expr provides an initial value, by means of an initialization expression, for the entity specified by the immediately preceding name.

--- Fortran 95 ---
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 object, 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.

Fortran 95

If the entity you are declaring is a derived type component, and initialization_expr or NULL() is specified, the derived type has default initialization.

End of Fortran 95

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 ALLOCATABLE or 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.

IBM Extension

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.

End of IBM Extension

Examples

```fortran
MODULE INT
  INTEGER, DIMENSION(3) :: A,B,C
  INTEGER :: X=234,Y=678
END MODULE INT
```

Related information

- "Integer" on page 16
- "Initialization expressions" on page 85
- "How type is determined" on page 51, for details on the implicit typing rules
- "Array declarators" on page 65
- "Automatic objects" on page 16
- "Storage classes for variables" on page 59
- "DATA" on page 273, for details on initial values
**INTENT**

**Purpose**

The INTENT attribute specifies the intended use of dummy arguments.

**Syntax**

```
<!-INTENT—(IN)--> dummy_arg_name_list ©<INOUT—OUT—IN>
```

*dummy_arg_name* is the name of a dummy argument, which cannot be a dummy procedure.

**Rules**

If you specify a nonpointer, nonallocatable dummy argument, the INTENT attribute will have the following characteristics:

- **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.

If you specify a pointer dummy argument, the INTENT attribute will have the following characteristics:

- **INTENT(IN)** specifies that during the execution of the procedure, the association status of the pointer dummy argument cannot be changed, except if the target of the pointer is deallocated. If the target of the pointer is deallocated, the association status of the pointer dummy argument becomes undefined.
  
  You cannot use an INTENT(IN) pointer dummy argument as a pointer object in a pointer assignment statement. You cannot allocate, deallocate, or nullify an INTENT(IN) pointer dummy argument.

  You cannot specify an INTENT(IN) pointer dummy argument as an actual argument to a procedure if the associated dummy argument is a pointer with INTENT(OUT) or INTENT(INOUT) attribute.

- **INTENT(OUT)** specifies that at the execution of the procedure, the association status of the pointer dummy argument is undefined.
- **INTENT(INOUT)** specifies that the dummy argument can both receive and return data to the invoking subprogram.

If you specify an allocatable dummy argument, the INTENT attribute will have the following characteristics:

- **INTENT(IN)** specifies that during the execution of the procedure, the allocation status of the dummy argument cannot be changed, and it must not be redefined or become undefined.
- **INTENT(OUT)** specifies that at the execution of the procedure, if the associated actual argument is currently allocated it will be deallocated.
• **INTENT(INOUT)** specifies that the dummy argument can both receive and return data to the invoking subprogram.

If you do not specify the **INTENT** attribute for a pointer or allocatable dummy argument, its use is subject to the limitations and restrictions of the associated actual argument.

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**).

### Attributes compatible with the **INTENT** attribute

<table>
<thead>
<tr>
<th>ALLOCATABLE</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIMENSION</td>
<td>TARGET</td>
</tr>
<tr>
<td>OPTIONAL</td>
<td>VALUE</td>
</tr>
<tr>
<td></td>
<td>VOLATILE</td>
</tr>
</tbody>
</table>

You must not specify the **VALUE** attribute for a dummy argument with an intent of **OUT** or **INOUT**.

---

**IBM Extension**

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 has no intent specified. This constraint does not apply to the %REF built-in function.

---

### Examples

```fortran
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
    B=C**2        ! Valid redefinition of B
END SUBROUTINE
```

### Related information

- “Intent of dummy arguments” on page 162
- “Argument association” on page 160
 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.

Syntax

```
 generalized operator ( defined_operator )
```

generic_spec

is

```
 generic_name
     OPERATOR --- defined_operator ---
     ASSIGNMENT --- = ---
```

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.

Fortran 95

If a generic_spec appears in an INTERFACE statement, it must match the generic_spec in the corresponding END INTERFACE statement.

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.

End of Fortran 95

An INTERFACE statement without a generic_spec can match any END INTERFACE statement, with or without a generic_spec.

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**

```fortran
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 140
- “Extended intrinsic and defined operations” on page 96
- “Defined operators” on page 146
- “Defined assignment” on page 147
- “FUNCTION” on page 317
- “SUBROUTINE” on page 401
- “MODULE PROCEDURE” on page 352
- “Procedure references” on page 155
- “Unambiguous generic procedure references” on page 144, 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.

**Syntax**
\texttt{\texttt{\textbf{INSTRINIC} }\texttt{:: name\_list}}

\textit{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 \texttt{INSTRINIC} attribute. Generic names can have the \texttt{INSTRINIC} attribute, but you cannot pass them as arguments unless they are also specific names.

A generic or specific procedure that has the \texttt{INSTRINIC} attribute keeps its generic or specific properties.

A generic intrinsic procedure that has the \texttt{INSTRINIC} 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 INSTRINIC attribute**

- \texttt{PRIVATE}
- \texttt{PUBLIC}

**Examples**

\begin{verbatim}
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
\end{verbatim}

**Related information**

- Generic and specific intrinsic procedures are listed in Chapter 13, “Intrinsic procedures,” on page 473. See this section to find out if a specific intrinsic name can be used as an actual argument.
- “Generic interface blocks” on page 144
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.

Syntax

```
LOGICAL kind_selector attr_spec_list entity_decl_list
```

where:

<table>
<thead>
<tr>
<th>attr_spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALLOCATABLE</td>
</tr>
<tr>
<td>AUTOMATIC</td>
</tr>
<tr>
<td>BIND</td>
</tr>
<tr>
<td>DIMENSION (array_spec)</td>
</tr>
<tr>
<td>EXTERNAL</td>
</tr>
<tr>
<td>INTENT (intent_spec)</td>
</tr>
<tr>
<td>INTRINSIC</td>
</tr>
<tr>
<td>OPTIONAL</td>
</tr>
<tr>
<td>PARAMETER</td>
</tr>
<tr>
<td>POINTER</td>
</tr>
<tr>
<td>PRIVATE</td>
</tr>
<tr>
<td>PUBLIC</td>
</tr>
<tr>
<td>SAVE</td>
</tr>
<tr>
<td>STATIC</td>
</tr>
<tr>
<td>TARGET</td>
</tr>
<tr>
<td>VOLATILE</td>
</tr>
</tbody>
</table>

346  XL Fortran Language Reference
kind_selector

\[
\begin{align*}
& ( \text{KIND} = \text{int_initialization_expr} ) \\
& \times \text{int_literal_constant} \\
\end{align*}
\]

Notes:
1 IBM Extension.

---

IBM Extension

specifies the length of logical entities: 1, 2, 4 or 8. int_literal_constant cannot specify a kind type parameter.

End of IBM Extension

---

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. Use the double colon separator when you specify attributes, =initialization_expr, or => NULL().

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 with an implicit interface.

**IBM Extension**

*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.

**End of IBM Extension**

**IBM Extension**

*initial_value* provides an initial value for the entity specified by the immediately preceding name

**End of IBM Extension**

*initialization_expr* provides an initial value, by means of an initialization expression, for the entity specified by the immediately preceding name

**Fortran 95**
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.

**Fortran 95**

If the entity you are declaring is a derived type component, and *initialization_expr* or **NULL()** is specified, the derived type has default initialization.

---

If 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 **ALLOCATABLE** or **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.

---

**IBM Extension**

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**

```fortran
LOGICAL, ALLOCATABLE :: L(:,:)
LOGICAL :: Z=.TRUE.
```

**Related information**

- “Logical” on page 24
- “Initialization expressions” on page 85
- “How type is determined” on page 51, for details on the implicit typing rules
- “Array declarators” on page 65
- “Automatic objects” on page 16
- “Storage classes for variables” on page 59
- “DATA” on page 273, 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.

Syntax

```
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. A user-defined module 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
      DO 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 149
- “USE” on page 414
- “Use association” on page 135
- “END” on page 292 for details on the END MODULE statement
- “PRIVATE” on page 369
- “PROTECTED” on page 374
- “PUBLIC” on page 376
MODULE PROCEDURE

Purpose
The MODULE PROCEDURE statement lists those module procedures that have a generic interface.

Syntax

```
MODULE PROCEDURE—procedure_name_list
```

Rules

Fortran 95

The MODULE PROCEDURE statement can appear anywhere among the interface bodies in an interface block that has a generic specification.

```
MODULE PROCEDURE procedure_name_list
```

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 accessible 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
    PRINT *, "In S1"
  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
SUBROUTINE SS1(IARG,JARG)
  PRINT *, "In SS1"
END SUBROUTINE SS1
```
NAMELIST

Purpose

The NAMELIST statement specifies one or more lists of names for use in READ, WRITE and PRINT statements.

Syntax

```latex
\text{\textbackslash NAMELIST} \text{\textbackslash Nname} \text{\textbackslash \text{-variable\_name\_list}}
```

\textit{Nname} is a namelist group name

\textit{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 object, 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.

\textit{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.

\textit{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.

\textit{Nname} can be specified in more than one NAMELIST statement in the scoping unit, and more than once in each NAMELIST statement. The \textit{variable\_name\_list} following each successive appearance of the same \textit{Nname} in a scoping unit is treated as the continuation of the list for that \textit{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 226
- Setting Run-time Options in the XL Fortran Compiler Reference

NULLIFY

Purpose

The NULLIFY statement causes pointers to become disassociated.

Syntax

```
pointer_object
```

is a pointer variable name or structure component

Rules

A `pointer_object` must be definable and have the POINTER attribute.

A `pointer_object` must not depend on the value, bounds, or association status of another `pointer_object` in the same NULLIFY statement.

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 112
- “Pointer association” on page 136
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.

Syntax

```
OPEN(open_list)
```

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 an 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.

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, DIRECT or STREAM. If ACCESS= is DIRECT, RECL= must be specified. If ACCESS= is STREAM, RECL= must not be specified.

SEQUENTIAL is the default, for which RECL= is optional

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.
IBM Extension

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.

IBM Extension

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.

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.

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.

IBM Extension

char_expr is a scalar character expression whose value, when any trailing
blanks are removed, is a valid Linux 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 Linux 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).

\[\text{FORM} = \text{char_expr}\]

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 you connect the file for sequential access, FORMATTED is the default. If you connect the file for direct access or stream access, UNFORMATTED is the default.

\[\text{IOMSG} = \text{iomsg\_variable}\]

is an input/output status specifier that specifies the message returned by the input/output operation. iomsg_variable is a scalar default character variable. It must not be a use-associated nonpointer protected variable. When the input/output statement containing this specifier finishes execution, iomsg_variable is defined as follows:

- If an error, end-of-file, or end-of-record condition occurs, the variable is assigned an explanatory message as if by assignment.
- If no such condition occurs, the value of the variable is unchanged.

\[\text{IOSTAT} = \text{ios}\]

is an input/output status specifier that specifies the status of the input/output operation. ios is a scalar variable. 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.

\[\text{PAD} = \text{char\_expr}\]

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.

**IBM Extension**

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.

\[\text{End of IBM Extension} \]
POSITION= char_expr
specifies the file position for a file connected for sequential or stream 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.

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 an 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.

<table>
<thead>
<tr>
<th>IBM Extension</th>
</tr>
</thead>
</table>

If RECL= is omitted when a file is being connected for sequential access in 32-bit, the length is 2**31-1, minus the record terminator. For a formatted sequential file in 32-bit, the default record length is 2**31-2. For an unformatted file that can be accessed in 32-bit, the default record length is 2**31-9.

For a file that cannot be accessed randomly in 32-bit, the default length is 2**15 (32,768).

If RECL= is omitted when a file is being connected for sequential access in 64-bit, the length is 2**63-1, minus the record terminator. For a formatted sequential file in 64-bit, the default record length is 2**63-2. For an unformatted file in 64-bit, the default record length is 2**63-17 when the UWIDTH run-time option is set to 64.

| End of IBM Extension |

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.

**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.

---

**IBM Extension**

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.

---

**End of IBM Extension**

---

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

• “Units” on page 180
• Item 3 under Appendix A, “Compatibility across standards,” on page 743
• Chapter 8, “XL Fortran Input/Output,” on page 177
• Setting Run-time Options in the XL Fortran Compiler Reference
• -qposition Option in the XL Fortran Compiler Reference
• -qxl77 Option in the XL Fortran Compiler Reference
• “CLOSE” on page 260
• “READ” on page 377
• “WRITE” on page 425

OPTIONAL

Purpose

The OPTIONAL attribute specifies that a dummy argument need not be associated with an actual argument in a reference to the procedure.

Syntax

```fortran
>>>OPTIONAL<<< dummy_arg_name_list
```
Rules

A procedure that has an optional dummy argument must have an explicit interface in any scope in which the procedure is referenced.

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

| ALLOCATABLE | INTENT | VALUE |
| DIMENSION | POINTER | VOLATILE |
| EXTERNAL | TARGET |

Examples

```fortran
SUBROUTINE SUB (X,Y)
  INTERFACE
    SUBROUTINE SUB2 (A,B)
      OPTIONAL :: B
    END SUBROUTINE
  END INTERFACE
  OPTIONAL :: Y
  IF (PRESENT(Y)) THEN
    X = X + Y
  ! Reference to Y conditional
  ELSE
    ! on its presence
  ENDIF
  CALL SUB2(X,Y)
END SUBROUTINE

SUBROUTINE SUB2 (A,B)
  OPTIONAL :: B
  IF (PRESENT(B)) THEN
    B = B * A
  ! B and Y are argument associated,
  ! even if Y is not present, in
  PRINT*, B
  ! which case, B is also not present
```
ELSE
   A = A**2
PRINT*, A
ENDIF
END SUBROUTINE

Related information
- “Optional dummy arguments” on page 163
- “Interface concepts” on page 139
- “PRESENT(A)” on page 569
- “Dummy arguments” on page 159

PARAMETER

Purpose
The PARAMETER attribute specifies names for constants.

Syntax

```
PARAMETER ( constant_name = init_expr )
```

*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.

<table>
<thead>
<tr>
<th>Attributes compatible with the PARAMETER attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIMENSION</td>
</tr>
</tbody>
</table>

362 XL Fortran Language Reference
Examples

```fortran
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 = ' '
```

Related information

- “Initialization expressions” on page 85
- “Data objects” on page 15

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.

Syntax

```
PAUSE

char_constant
digit_string
```

*char_constant*

is a scalar character constant that is not a Hollerith constant

*digit_string*

is a string of one to five digits

Rules

**IBM Extension**

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.

**End of IBM Extension**

**Fortran 95**

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 746

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.

Syntax


defered_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 or NAMELIST statement.

Pointer variables can appear in common blocks and block data program units.

IBM Extension

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 -qsavc 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.

End of IBM Extension
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
- INTENT
- OPTIONAL
- PRIVATE
- PROTECTED
- 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

**Example 1:**

```fortran
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
PRINT *, FUNC()
CONTAINS
  REAL FUNCTION FUNC()
  POINTER :: FUNC
      ! Function result is a pointer
  ...
  ...
  END FUNCTION
END
```

**IBM Extension**

**Example 2:** Fortran 90 pointers and threadsafing

```fortran
FUNCTION MYFUNC(ARG)
      ! MYPTR is thread-specific.
INTEGER, POINTER :: MYPTR
ALLOCATE(MYPTR)
MYPTR = ARG
ANYVAR = MYPTR
END FUNCTION
```

---

### Related information

- “Pointer assignment” on page 112
- “TARGET” on page 403
- “ALLOCATED(X)” on page 485
- “DEALLOCATE” on page 276
- “Pointer association” on page 136
The integer POINTER statement specifies that the value of the variable \textit{int\_pointer} is to be used as the address for any reference to \textit{pointee}.

The name of this statement has been changed from \texttt{POINTER} to integer \texttt{POINTER} to distinguish it from the Fortran 90 \texttt{POINTER} statement.

**Syntax**

\[
\text{\texttt{POINTER}}(\text{int\_pointer}, \text{pointee})
\]

- \textit{int\_pointer} is the name of an integer pointer variable
- \textit{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 \texttt{INTEGER(4)} in 32-bit mode and type \texttt{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 \texttt{POINTER} statement is called a pointee array. You can dimension a pointee array in a type declaration statement, a \texttt{DIMENSION} statement, or in the integer \texttt{POINTER} statement itself.

If you specify the \texttt{-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 \texttt{-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, though a pointee can have nonconstant bounds or lengths.
• 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:
  – ALLOCATABLE
  – DIMENSION
  – EXTERNAL
  – INTRINSIC
  – PARAMETER
  – POINTER
  – TARGET
• An integer pointer cannot appear as a NAMELIST group name.
• An integer pointer cannot be a procedure.

Examples

```fortran
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 137
• “LOC(X)” on page 542
• -qddim Option in the XL Fortran Compiler Reference

---

PRINT

Purpose

The PRINT statement is a data transfer output statement.

Syntax
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 object must be allocated. A derived-type object cannot have any ultimate component that is inaccessible to 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 369.

Fortran 2003 Standard

An expression that is an output_item cannot have a value that is a procedure pointer.

End of Fortran 2003 Standard

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

Fortran 95 does not permit assigning of a statement label.

End of Fortran 95

- 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 a previously defined namelist.

Specifying the `-qport=typestmt` compiler option enables the TYPE statement which has identical functionality to the PRINT statement.

**Implied-DO List**

```plaintext

((do_object_list , do_variable = arith_expr1, arith_expr2)
 [ , arith_expr3 ])
```

*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**

```plaintext
PRINT 10, A,B,C
10 FORMAT (E4.2,G3.2E1,B3)
```

**Related information**

- Chapter 8, “XL Fortran Input/Output,” on page 177
- Chapter 9, “Input/Output formatting,” on page 195
- See the XL Fortran Compiler Reference for more information on `-qport=typestmt`
- “Deleted features” on page 746

---

**PRIVATE**

**Purpose**

The PRIVATE attribute specifies that a module entity is not accessible outside the module through use association.

**Syntax**
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 a PRIVATE statement without an access_id_list, 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. The accessibility of a derived type does not affect, and is not affected by, the accessibility of its components.

A namelist group must be private if it contains any object that is private or contains 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
- PROTECTED
- SAVE
- STATIC
- TARGET
- VOLATILE

Examples

```fortran
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
```

XL Fortran Language Reference
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 28
- “Modules” on page 149
- “PROTECTED” on page 374
- “PUBLIC” on page 376

PROCEDURE

Fortran 2003 Standard

Purpose
A PROCEDURE statement declares a dummy procedure, an external procedure, or a procedure pointer. It specifies the EXTERNAL attribute for these entities.

Syntax

```plaintext
PROCEDURE (procedure_interface)
:: procedure_attribute_list
:: procedure_entity_name
    => null_init
```

where:
- `procedure_interface` is a declaration type specifier or the name of a procedure that has an explicit interface.
- `procedure_attribute_list` is a list of attributes from the following list:
  - BIND
  - INTENT(intent_spec)
  - NOPASS
  - OPTIONAL
  - POINTER
Notes:
1. Does not apply to a procedure pointer component of a derived type
2. Applies only to a procedure pointer component of a derived type

:: is the double colon separator. It is required if attributes are specified.

procedure_entity_name is the name of the procedure or procedure pointer that is being declared.

null_init nullifies a procedure pointer.

Rules

If procedure_interface is the name of a procedure or procedure pointer that has an explicit interface, the declared procedures or procedure pointers have this explicit interface. The referenced procedure or procedure pointer must already be declared. The referenced procedure must not be an intrinsic procedure and its name cannot be the same as a keyword that specifies an intrinsic type. If the referenced procedure is an elemental procedure, the procedure entity names must consist of external procedures.

If procedure_interface is a declaration type specifier, the declared procedures or procedure pointers are functions with an implicit interface and the specified result type. If these functions are external functions, the function definitions must specify the same result type and type parameters.

If no procedure_interface is specified, the PROCEDURE statement specifies that the declared procedures or procedure pointers are either subroutines or functions. If they are functions, the implicit type rule applies to the type of the function.

If you specify procedure language binding using the BIND attribute, procedure_interface must be the name of a procedure or procedure pointer that is declared with procedure language binding.

If procedure language binding with NAME= is specified, the procedure entity name must consist of only one procedure entity name. This procedure must not be a dummy procedure or have the POINTER attribute.

If OPTIONAL is specified, the declared procedures or procedure pointers must be dummy procedures or procedure pointers.

You can only specify PUBLIC or PRIVATE if the statement appears in the specification part of a module.

If INTENT, SAVE, or null_init is specified, the declared entities must have the POINTER attribute.

If null_init is used, it specifies that the initial association status of the corresponding procedure pointer is disassociated. It also implies the SAVE attribute, which can be reaffirmed by explicitly using the SAVE attribute in the procedure declaration statement or by a SAVE statement.
For procedure pointer declarations, you must specify the \texttt{POINTER} attribute. For declarations of procedure pointer components of derived types, you must specify the \texttt{NOPASS} attribute.

**Examples**

**Example 1**

The following example shows an external procedure declaration.

```fortran
CONTAINS
SUBROUTINE XXX(PSI)
  PROCEDURE (REAL) :: PSI
  REAL Y1
  Y1 = PSI()
END SUBROUTINE
END
```

**Example 2**

The following example shows a procedure pointer declaration and its use.

```fortran
PROGRAM PROC_PTR_EXAMPLE
  REAL :: R1
  INTEGER :: I1
  INTERFACE
    SUBROUTINE SUB(X)
      REAL, INTENT(IN) :: X
    END SUBROUTINE SUB
    FUNCTION REAL_FUNC(Y)
      REAL, INTENT(IN) :: Y
      REAL, REAL_FUNC
    END FUNCTION REAL_FUNC
  END INTERFACE
  PROCEDURE(SUB), POINTER :: PTR_TO_SUB
    ! with explicit interface
  PROCEDURE(REAL_FUNC), POINTER :: PTR_TO_REAL_FUNC => NULL()
    ! with explicit interface
  PROCEDURE(INTEGER), POINTER :: PTR_TO_INT
    ! with implicit interface
  PTR_TO_SUB => SUB
  PTR_TO_REAL_FUNC => REAL_FUNC
  CALL PTR_TO_SUB(1.0)
  R1 = PTR_TO_REAL_FUNC(2.0)
  I1 = PTR_TO_INT(M, N)
END PROGRAM PROC_PTR_EXAMPLE
```

**Related information**

- “\texttt{BIND}” on page 247
- Chapter 13, “Intrinsic procedures,” on page 473
- Chapter 14, “Hardware-specific intrinsic procedures,” on page 607
- “Program units, procedures, and subprograms” on page 138
- “Intrinsic procedures” on page 156
- “\texttt{INTERFACE}” on page 343
- “Procedure pointer assignment” on page 114
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.

Syntax

```
PROGRAM name
```

(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 148

PROTECTED

```
```

Fortran 2003 Standard

Purpose
The PROTECTED attribute allows greater control over the modification of module entities. A module procedure can only modify a protected module entity or its subobjects if the same module defines both the procedure and the entity.
Syntax

The **PROTECTED** attribute must only appear in the specification part of the module.

```
entity_declaration_list ::
```

*entity*  A named variable not in a common block.

Rules

If you specify that an object declared by an **EQUIVALENCE** statement has the **PROTECTED** attribute, all objects specified in that **EQUIVALENCE** statement must have the **PROTECTED** attribute.

A nonpointer object with the **PROTECTED** attribute accessed through use association, is not definable.

You must not specify the **PROTECTED** attribute for **integer pointers**.

A pointer object with the **PROTECTED** attribute accessed through use association, must not appear as any of the following:

- As a pointer object in a **NULLIFY** statement or **POINTER** assignment statement
- As an allocatable object in an **ALLOCATE** or **DEALLOCATE** statement.
- As an actual argument in reference to a procedure, if the associated dummy argument is a pointer with the **INTENT(INOUT)** or **INTENT(OUT)** attribute.

### Attributes compatible with the **PROTECTED** attribute

- **ALLOCATABLE**
- **AUTOMATIC**
- **DIMENSION**
- **INTENT**
- **OPTIONAL**
- **PRIVATE**
- **PUBLIC**
- **POINTER**
- **SAVE**
- **STATIC**
- **TARGET**
- **VOLATILE**

Examples

In the following example, the values of both *age* and *val* can only be modified by subroutines in the module in which they are declared:

```fortran
module mod1
  integer, protected :: val
  integer :: age
  protected :: age
  contains
    subroutine set_val(arg)
      integer arg
      val = arg
    end subroutine
    subroutine set_age(arg)
      integer arg
      age = arg
    end subroutine
end module
```

Chapter 10. Statements and attributes  375
program dt_init01
    use mod1
    implicit none
    integer :: value, his_age
    call set_val(88)
    call set_age(38)
    value = val
    his_age = age
    print *, value, his_age
end program

Related information

"Modules" on page 149
"PRIVATE" on page 369
"PUBLIC"

--------- End of Fortran 2003 Standard ---------

---

PUBLIC

Purpose

The PUBLIC attribute specifies that a module entity can be accessed by other program units through use association.

Syntax

```
PUBLIC access_id_list
```

- `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 a PRIVATE statement without an access_id_list, 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.

--- IBM Extension ---

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.

--- End of IBM Extension ---

Attributes compatible with the PUBLIC attribute

Notes:
1. Fortran 2003 Standard
   
   - ALLOCATABLE
   - DIMENSION
   - EXTERNAL
   - INTRINSIC
   - PARAMETER
   - POINTER
   - PROTECTED (1)
   - SAVE
   - TARGET
   - VOLATILE (1)

Examples

```fortran
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 INTERFACE GEN
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 369
- “PROTECTED” on page 374
- “Modules” on page 149

--- READ ---

Purpose

The READ statement is the data transfer input statement.
### Syntax

```
READ(name, format, input_item_list, io_control_list)
```

- **format** is a format identifier, described below under `FMT=format`. In addition, it cannot be a Hollerith constant.
- **name** is an 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.
  - An implied-DO list, as described under "Implied-DO List" on page 383.
  - 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 object 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 383.

---

**Fortran 2003 Standard**

An `input_item` must not be a procedure pointer.

---

**IBM Extension**

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.

\[ \text{[FMT=} \text{]} \text{format} \]

is a format specifier that specifies the format to be used in the input operation. \text{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.

\[ \text{Fortran 95} \]

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 \[\text{“FORMAT” on page 315}\] can be used between the parentheses. Blank characters can precede the left parenthesis or follow the right parenthesis. If \text{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 \text{namelist} specifier that specifies a previously-defined namelist.

If the optional characters FMT= are omitted, \text{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.

ADVANCE= char_expr

is an advance specifier that determines whether nonadvancing input occurs for this statement. \text{char_expr} is a scalar character expression that must evaluate to \text{YES} or \text{NO}. If \text{NO} is specified, nonadvancing input occurs. If \text{YES} is specified, advancing, formatted sequential or stream input occurs. The default value is \text{YES}. ADVANCE= can be specified only in a formatted
sequential or formatted stream READ statement with an explicit format specification that does not specify an internal file unit specifier.

**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.

**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.

---

**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.

---

**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.

---

**IBM Extension**

**ID= integer_variable**

indicates that the data transfer is to be done asynchronously. The integer_variable is an integer variable. 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, sequential unformatted or stream 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**.

End of IBM Extension

---

**Fortran 2003 Standard**

**IOMSG=** **iomsg_variable**

is an input/output status specifier that specifies the message returned by the input/output operation. **iomsg_variable** is a scalar default character variable. It must not be a use-associated nonpointer protected variable. When the input/output statement containing this specifier finishes execution, **iomsg_variable** is defined as follows:

- If an error, end-of-file, or end-of-record condition occurs, the variable is assigned an explanatory message as if by assignment.
- If no such condition occurs, the value of the variable is unchanged.

End of Fortran 2003 Standard

---

**IOSTAT=** **ios**

is an input/output status specifier that specifies the status of the input/output operation. **ios** is an integer variable. 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.

---

**Fortran 2003 Standard**

**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.

--- IBM Extension ---

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.

--- End of IBM Extension ---

[NML=] name

is a namelist specifier that specifies a previously-defined namelist. 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.

--- IBM Extension ---

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 an integer variable. 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.

--- End of IBM Extension ---

POS=integer_expr

is an integer expression greater than 0. POS= specifies the file position of the file storage unit to be read in a file connected for stream access. You must not use this specifier for a file that cannot be positioned.

REC= integer_expr

is a record specifier that specifies the number of the record to be read.

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. You must not specify REC= in data transfer statements that specify a unit connected for stream access, or use the POS= 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 an integer variable. Blanks that are inserted as padding are not included in the count.

**Implied-DO List**

\[
\begin{align*}
\text{do_object} & \quad \text{is an output list item} \\
\text{do_variable} & \quad \text{is a named scalar variable of type integer or real} \\
\text{arith_expr1, arith_expr2, and arith_expr3} & \quad \text{are scalar numeric expressions}
\end{align*}
\]

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.

--- IBM Extension

If a NUM= specifier is present, neither a format specifier nor a namelist specifier can be present.

--- End of IBM Extension

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.

---

**IBM Extension**

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**

```fortran
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**

- “Asynchronous Input/Output” on page 183
- Implementation details of XL Fortran Input/Output in the XL Fortran Optimization and Programming Guide
- “Conditions and IOSTAT values” on page 186
- “WRITE” on page 425
- “WAIT” on page 421
- Chapter 8, “XL Fortran Input/Output,” on page 177
- Setting run-time options in the XL Fortran Compiler Reference
- “Deleted features” on page 746
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.

Syntax

\[
\begin{align*}
\text{REAL} & \quad \text{kind_selector} \quad :: \quad \text{attr_spec_list} \quad ::= \quad \text{entity_decl_list} \\
\end{align*}
\]

where:

\[
\text{attr_spec} \quad ::= \\
\text{ALLOCATABLE} \\
\text{AUTOMATIC} \\
\text{BIND} \\
\text{DIMENSION} \quad (\text{array_spec}) \\
\text{EXTERNAL} \\
\text{INTENT} \quad (\text{intent_spec}) \\
\text{INTRINSIC} \\
\text{OPTIONAL} \\
\text{PARAMETER} \\
\text{PRIVATE} \\
\text{PUBLIC} \\
\text{SAVE} \\
\text{STATIC} \\
\text{TARGAET} \\
\text{VOLATILE}
\]

\[
\text{kind_selector} \quad ::= \\
\left( \frac{\text{KIND} = \text{int_initialization_expr}}{\text{int_literal_constant}} \right) \quad (1)
\]

Notes:

1 IBM Extension.

IBM Extension

specifies the length of real entities: 4, 8 or 16. \text{int_literal_constant} cannot specify a kind type parameter.

End of IBM Extension
\[a\]

is an object name or function name. \(\text{array_spec}\) cannot be specified for a function name with an implicit interface.

**IBM Extension**

\(\text{len}\) overrides the length as specified in \(\text{kind_selector}\), and cannot specify a kind type parameter. The entity length must be an integer literal constant that represents one of the permissible length specifications.

**End of IBM Extension**

**IBM Extension**

\(\text{initial_value}\)

provides an initial value for the entity specified by the immediately
preceding name.

**End of IBM Extension**

`initialization_expr` provides an initial value, by means of an initialization expression, for the entity specified by the immediately preceding name.

**Fortran 95**

`=> NULL()` provides the initial value for the pointer object.

**End of Fortran 95**

**Rules**

**Fortran 95**

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.

**End of Fortran 95**

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 object, a function result, an object in a blank common block, 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.

**IBM Extension**

The object also may be initialized 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 ALLOCATABLE or 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.

IBM Extension

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 18
- “Initialization expressions” on page 85
- “How type is determined” on page 51, for details on the implicit typing rules
- “Array declarators” on page 65
- “Automatic objects” on page 16
- “Storage classes for variables” on page 59
- “DATA” on page 273, for details on initial values

RECORD

IBM Extension

Purpose

The RECORD statement is a special form of type declaration statement. Unlike other type declaration statements, attributes for entities declared on the RECORD statement cannot be specified on the statement itself.

Syntax

record_stmt:

"""RECORD / type_name / record_obj_dcl_list :: record_obj_dcl:

"""RECORD / type_name / record_obj_dcl_list:

record_obj_dcl:

"""record_object_name (-array_spec-):
where *type_name* must be the name of a derived type that is accessible in the scoping unit.

**Rules**

Entities can not be initialized in a **RECORD** statement.

A *record_stmt* declares an entity to be of the derived type, specified by the *type_name* that most immediately precedes it.

The **RECORD** keyword must not appear as the *type_spec* of an **IMPLICIT** or **FUNCTION** statement.

---

**Fortran 2003 Standard**

A derived type with the **BIND** attribute must not be specified in a **RECORD** statement.

---

**Examples**

In the following example, a **RECORD** statement is used to declare a derived type variable.

```fortran
STRUCTURE /S/
  INTEGER I
END STRUCTURE
STRUCTURE /DT/
  INTEGER I
END STRUCTURE
RECORD/DT/REC1,REC2,/S/REC3,REC4
```

**Related information**

- For further information on record structures and derived types, see "Derived types" on page 28

---

**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.
Syntax

![RETURN](1)

Notes:
1. Real or complex expressions are an IBM Extension.

*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 undefined” 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 168
- “Actual argument specification” on page 157 for a description of alternate return points
- “Events causing undefined” 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. For stream access, the **REWIND** statement positions a file at its initial point.

Syntax

```fortran
REWIND u (position_list)
```

- **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 an 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**.
  - **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.

**Fortran 2003 Standard**

- **IOMSG=** iomsg_variable
  - **iomsg_variable** is an input/output status specifier that specifies the message returned by the input/output operation. **iomsg_variable** is a scalar default character variable. It must not be a use-associated nonpointer protected variable. When the input/output statement containing this specifier finishes execution, **iomsg_variable** is defined as follows:
    - If an error, end-of-file, or end-of-record condition occurs, the variable is assigned an explanatory message as if by assignment.
    - If no such condition occurs, the value of the variable is unchanged.

End of **Fortran 2003 Standard**
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.

Syntax
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.

IBM Extension

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 Pthreads library module in the XL Fortran Optimization and Programming Guide for more information. The lock_name attribute on the CRITICAL directive also provides the ability to serialize access to data. See CRITICAL/END CRITICAL in the XL Fortran Optimization and Programming Guide for more information. The THREADLOCAL directive ensures that common blocks are local to each thread. See THREADLOCAL in the XL Fortran Optimization and Programming Guide for more information.

End of IBM Extension
Attributes compatible with the SAVE attribute

- ALLOCATABLE
- DIMENSION
- POINTER
- PRIVATE
- PROTECTED
- 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 262
- THREADLOCAL in the XL Fortran Optimization and Programming Guide
- “Definition status of variables” on page 52
- “Storage classes for variables” on page 59
- Item 2 under Appendix A, “Compatibility across standards,” on page 743

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.

Syntax

```
SELECT CASE (case_expr)
    case_construct_name:
        statements
    END CASE
```

- `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.

---

IBM Extension

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

- “SELECT CASE construct” on page 125
- “CASE” on page 254
- “END (Construct)” on page 293, 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.

Syntax

```fortran
SEQUENCE
```

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 28
- “Derived Type” on page 278
- “END TYPE” on page 297

Statement Function

Purpose

A statement function defines a function in a single statement.

Syntax

```
name(dummy_argument_list) = scalar_expression
```

- `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.

\textbf{IBM Extension}

With XL Fortran, the scalar expression can also reference a structure constructor.

\textbf{End of IBM Extension}

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.

\textbf{Examples}

\begin{verbatim}
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)
\end{verbatim}
STATIC

IBM Extension

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.

Syntax

```
STATIC :: stat_variable [:: initial_value_list] /...
```

- `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 273.

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 [XL Fortran Compiler Reference](#) 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 [Pthreads library module](#) in the [XL Fortran Optimization and Programming Guide](#) for more information. The `lock_name` attribute on the CRITICAL directive also provides the ability to serialize
access to data. See `CRITICAL/END CRITICAL` in the `XL Fortran Optimization and Programming Guide` for more information. The `THREADLOCAL` directive ensures that common blocks are local to each thread. See `THREADLOCAL` in the `XL Fortran Optimization and Programming Guide` for more information.

### Attributes compatible with the STATIC attribute

- `ALLOCATABLE`
- `DIMENSION`
- `POINTER`
- `PRIVATE`
- `PROTECTED`
- `SAVE`
- `TARGET`
- `VOLATILE`

### Examples

```fortran
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
    WRITE (*, *) ! Output on first call is 2
    WRITE (*, *) ! Output on second call is 3
  END SUBROUTINE
END
```

### Related information

- "Storage classes for variables" on page 59
- "COMMON" on page 262
- `THREADLOCAL` in the `XL Fortran Optimization and Programming Guide`

---

**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.

### Syntax

```fortran
STOP [char_constant] [digit_string]
```
char_constant
  is a scalar character constant that is not a Hollerith constant

digit_string
  is a string of one through five digits

Rules

________________________________________________________________________
IBM Extension

If neither char_constant nor digit_string are specified, nothing is printed to standard error (unit 0).

________________________________________________________________________
End of IBM Extension

A STOP statement cannot terminate the range of a DO or DO WHILE construct.

________________________________________________________________________
IBM Extension

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 ?.

________________________________________________________________________
End of IBM Extension

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.

Syntax

```
  SUBROUTINE name [prefix, ] dummy_argument_list [ ]

  BIND(C, NAME = binding_label)
```

prefix  is one of the following:
• **F95** ELEMENTAL **F95**
• **F95** PURE **F95**
• **RECURSIVE**

Note: type_spec is not permitted as a prefix in a subroutine.

name is the name of the subroutine subprogram

---

**Fortran 2003 Standard**

binding_label
a scalar character initialization expression

---

**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.

---

**Fortran 95**

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. For elemental procedures, the keyword **ELEMENTAL** must be specified. If the **ELEMENTAL** keyword is specified, the **RECURSIVE** keyword cannot be specified.

---

**IBM Extension**

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.
The **BIND** keyword implicitly or explicitly defines a binding label by which a procedure is accessed from the C programming language. A dummy argument cannot be zero-sized. A dummy argument for a procedure with the **BIND** attribute must have interoperable types and type parameters, and cannot have the **ALLOCATABLE, OPTIONAL, or POINTER** attribute.

The **BIND** attribute must not be specified for an internal procedure. If the **SUBROUTINE** statement appears as part of an interface body that describes a dummy procedure, the **NAME** specifier must not appear. An elemental procedure cannot have the **BIND** attribute.

---

**Examples**

```fortran
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 154
- “Dummy arguments” on page 159
- “Recursion” on page 171
- “CALL” on page 252
- “ENTRY” on page 300
- “Statement Function” on page 397
- “BIND” on page 247
- “RETURN” on page 390
- “Definition status of variables” on page 52
- “Pure procedures” on page 171
- `-qrecur` option in the **XL Fortran Compiler Reference**

---

**TARGET**

**Purpose**

Data objects with the **TARGET** attribute can be associated with pointers.

**Syntax**
Rules

If a data object has the TARGET attribute, then all of the data object’s nonpointer subobjects will also have the TARGET attribute.

A data object that does not have the TARGET attribute cannot be associated with an accessible pointer.

A target cannot appear in an EQUIVALENCE statement.

--- IBM Extension ---

A target cannot be an integer pointer or a pointee.

--- End of IBM Extension ---

Attributes compatible with the TARGET attribute

- ALLOCATABLE
- AUTOMATIC
- DIMENSION
- INTENT
- OPTIONAL
- PRIVATE
- PROTECTED
- PUBLIC
- SAVE
- STATIC
- VALUE
- 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(X)" on page 485
- "DEALLOCATE" on page 276
- "Pointer assignment" on page 112
- "Pointer association" on page 136

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.
Syntax

\[
\text{TYPE} \text{-(type_name)-} \text{entity_decl_list-}\|
\]

where:

\[
\begin{array}{|c|}
\hline
\text{attr_spec} \\
\hline
\text{ALLOCATABLE} \\
\text{AUTOMATIC} \\
\text{BIND} \\
\text{DIMENSION (array_spec)} \\
\text{EXTERNAL} \\
\text{INTENT (intent_spec)} \\
\text{INTRINSIC} \\
\text{OPTIONAL} \\
\text{PARAMETER} \\
\text{POINTER} \\
\text{PRIVATE} \\
\text{PUBLIC} \\
\text{SAVE} \\
\text{STATIC} \\
\text{TARGET} \\
\text{VOLATILE} \\
\hline
\end{array}
\]

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 \text{IN}, \text{OUT}, or \text{INOUT}

::

is the double colon separator. It is required if attributes are specified, \text{= initialization_expr} is used, or \text{=}NULL() appears as part of any entity_decl.

array_spec

is a list of dimension bounds

entity_decl
is an object name or function name. *array_spec* cannot be specified for a function with an implicit interface.

**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 object, function result, object in a blank common block, 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 pointer 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.

**Fortran 95**

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 `ALLOCTABLE` or `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.

---

**IBM Extension**

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**

```fortran
TYPE PEOPLE!
  INTEGER AGE
  CHARACTER*20 NAME
END TYPE PEOPLE
TYPE(PEOPLE) :: SMITH = PEOPLE(25,'John Smith')
END
```

**Related information**

- “Derived types” on page 28
- “Derived Type” on page 278
- “Initialization expressions” on page 85
- “How type is determined” on page 51, for details on the implicit typing rules
- “Array declarators” on page 65
- “Automatic objects” on page 16
Type Declaration

Purpose

A type declaration statement specifies the type, length, and attributes of objects and functions. Initial values can be assigned to objects.

Syntax

\[
\text{::= \[ attr_spec_list \] \text{entity_decl_list} \text{::= \ldots} }
\]

where:

- **type_spec** is any of the following:
  - **BYTE**
  - **CHARACTER** [char_selector]
  - **COMPLEX** [kind_selector]
  - **DOUBLE COMPLEX**
  - **DOUBLE PRECISION**
  - **REAL** [kind_selector]
  - **INTEGER** [kind_selector]
  - **VECTOR** [type_spec]

- **attr_spec** is any of the following:
  - **ALLOCATABLE**
  - **AUTOMATIC**
  - **BIND(C) NAME=binding_label**
  - **DIMENSION** [array_spec]
  - **EXTERNAL**
  - **INTENT** [intent_spec]
  - **POINTER**
  - **TARGET**
  - **EXTERNAL**
  - **PRIVATE**
  - **PROTECTED**
  - **SAVE**
  - **STATIC**
  - **PARAMETER**
  - **VALUE**
  - **INTRINSIC**
  - **OPTIONAL**
  - **PARAMETER**
  - **STATIC**
  - **VALUE**
  - **PROTECTED**
  - **VOLATILE**

- **kind_selector**

\[
\text{::= \[ \text{int_initialization_expr} \] \text{::= \ldots} }
\]

Notes:

1. IBM Extension.

**VECTOR(type_spec)** must specify **PIXEL REAL** of kind 4, or **INTEGER** or **UNSIGNED** of kind 1, 2, or 4.
represents one of the permissible length specifications for its associated type.

--- IBM Extension ---

`int_literal_constant` cannot specify a kind type parameter.

--- End of IBM Extension ---

`char_selector` specifies the character length

--- IBM Extension ---

In XL Fortran, this is the 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.

--- End of IBM Extension ---

`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. Use the double colon separator when you specify attributes, `=initialization_expr` or `=> NULL()`.
array_spec
is a list of dimension bounds.

entity_decl

```
(a
  * --char_length--
  (--array_spec--)
  (--array_spec--) * --char_length--
)
```

Notes:
1 IBM Extension
2 Fortran 95

*a* is an object name or function name. array_spec cannot be specified for a function with an implicit interface.

IBM Extension

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.

---------- End of IBM Extension

IBM Extension

initial_value
provides an initial value for the entity specified by the immediately preceding name.

---------- End of IBM Extension

initialization_expr
provides an initial value, by mean of an initialization expression, for the entity specified by the immediately preceding name.

---------- Fortran 95
provides the initial value for the pointer object.

End of Fortran 95

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.

End of Fortran 95

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 object, function result, object in a blank common block, 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.

---

**Fortran 95**

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 **ALLOCATABLE** or **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.

---

**IBM Extension**

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.

---

**End of IBM Extension**

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 15
- “Initialization expressions” on page 85
- “How type is determined” on page 51 for details on the implicit typing rules
- “Array declarators” on page 65
- “Automatic objects” on page 16
- “Storage classes for variables” on page 59
- “DATA” on page 273, 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.

Syntax

```
USE module_name (1)
    ,rename_list 
    ,ONLY:
    ,INTRINSIC
    ,NON_INTRINSIC
```

Notes:

1  Fortran 2003 Standard
 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.

IBM Extension

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.

End of IBM Extension

 Entities in the scoping unit become use-associated with the module entities, and the local entities have the attributes of the corresponding module entities.

Fortran 2003 Standard

By default, either an intrinsic module or a non-intrinsic module with the specified name is accessed. If both an intrinsic module and a non-intrinsic module have this name, the non-intrinsic module is accessed. If you specify INTRINSIC or NON_INTRINSIC, only an intrinsic module or only a non-intrinsic module can be accessed.

End of Fortran 2003 Standard

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.

The accessed entities have the attributes specified in the module, except that an entity may have a different accessibility attribute or it may have the **VOLATILE** attribute in the local scoping unit even if the associated module entity does not.

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**
```
MODULE A
  REAL :: X=5.0
END MODULE A
MODULE B
USE A
PRIVATE :: X
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
```

**Fortran 2003 Standard**

The following example is invalid:

Module mod1
```
use, intrinsic :: ieee_exceptions
end Module
```

Module mod2
```
use, non_intrinsic :: ieee_exceptions
end Module
```

Program invalid_example
```
use mod1
use mod2
! ERROR: a scoping unit must not access an
```
Related information

- “Modules” on page 149
- “PRIVATE” on page 369
- “VOLATILE” on page 419
- “PUBLIC” on page 376
- “Order of statements and execution sequence” on page 14

VALUE

Purpose

The VALUE attribute specifies an argument association between a dummy and an actual argument. This association allows you to pass the dummy argument with the value of the actual argument. This pass by value implementation from the Fortran 2003 Standard provides a standard conforming option to the %VAL built-in function.

An actual argument and the associated dummy argument can change independently. Changes to the value or definition status of the dummy argument do not affect the actual argument. A dummy argument with the VALUE attribute becomes associated with a temporary variable with an initial value identical to the value of the actual argument.

Syntax

```
VALUE dummy_argument_name_list
```

Rules

You must specify the VALUE attribute for dummy arguments only.

You must not use the %VAL or %REF built-in functions to reference a dummy argument with the VALUE attribute, or the associated actual argument.

A referenced procedure that has a dummy argument with the VALUE attribute must have an explicit interface.

A dummy argument with the VALUE attribute can be of character type if you omit the length parameter or specify it using an initialization expression with a value of 1.
You must not specify the **VALUE** attribute with the following:

- Arrays
- Derived types with **ALLOCATABLE** components
- Dummy procedures

### Attributes compatible with the **VALUE** attribute

- **INTENT(IN)**
- **OPTIONAL**
- **TARGET**

If a dummy argument has both the **VALUE** and **TARGET** attributes, any pointers associated with that dummy argument become undefined after the execution of the procedure.

### Examples

Program validexm1
```
integer :: x = 10, y = 20
print *, 'before calling: ', x, y
call intersub(x, y)
print *, 'after calling: ', x, y

contains
subroutine intersub(x,y)
   integer, value :: x
   integer y
   x = x + y
   y = x*y
   print *, 'in subroutine after changing: ', x, y
end subroutine
end program validexm1
```

Expected output:
```
before calling: 10 20
in subroutine after changing: 30 600
after calling: 10 600
```

### Related information

For more information, see the `%VAL` built-in function.

---

**VECTOR**

**IBM Extension**

**Purpose**

A **VECTOR** type declaration statement specifies that one or more entities have a **vector type**.

**Syntax**

You can declare a vector using `VECTOR(type_spec)` as part of a type declaration statement. The **type declaration statement** contains the complete syntax for declaring a vector data type. In a `VECTOR(type_spec)`, `type_spec` must specify
VIRTUAL

IBM Extension

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.

Syntax

```
VIRTUAL array_declarator_list
```

Rules

IBM Extension

You can specify arrays with a maximum of 20 dimensions

End of IBM Extension

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,*), DATA(0:9,-5:4,10) ! assumed-size array

Related information

- Chapter 4, “Array concepts,” on page 63
- “DIMENSION” on page 279

End of IBM Extension

VOLATILE

IBM Extension

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.
Syntax

```
VOLATILE :: variable_name
             /--common_block_name--/
             derived_type_name
```

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
- PROTECTED
- 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

- "Direct access" on page 179

---

WAIT

IBM Extension

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.

Syntax

```
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:

**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 \texttt{WAIT} statement without the \texttt{DONE=} specifier, or a \texttt{WAIT} statement that sets the \texttt{logical_variable} value to true, is the matching \texttt{WAIT} statement to the data transfer statement identified by the same \texttt{ID=} value.

\texttt{END=} \texttt{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 \texttt{IOSTAT=} specifier, if present, is assigned a negative value, and the \texttt{NUM=} specifier, if present, is assigned an integer value. Coding the \texttt{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 \texttt{stmt\_label} defined for the \texttt{END=} specifier of the asynchronous data transfer statement need not be identical to the \texttt{stmt\_label} defined for the \texttt{END=} specifier of the matching \texttt{WAIT} statement.

\texttt{ERR=} \texttt{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 \texttt{ERR=} specifier suppresses error messages.

The \texttt{stmt\_label} defined for the \texttt{ERR=} specifier of the asynchronous data transfer statement need not be identical to the \texttt{stmt\_label} defined for the \texttt{ERR=} specifier of the matching \texttt{WAIT} statement.

\texttt{ID=} \texttt{integer\_expr}

indicates the data transfer with which this \texttt{WAIT} statement is identified. The \texttt{integer\_expr} is an integer expression of type \texttt{INTEGER(4)} or default integer. To initiate an asynchronous data transfer, the \texttt{ID=} specifier is used on a \texttt{READ} or \texttt{WRITE} statement.

\texttt{IOMSG=} \texttt{iomsg\_variable}

is an input/output status specifier that specifies the message returned by the input/output operation. \texttt{iomsg\_variable} is a scalar default character variable. It must not be a use-associated nonpointer protected variable. When the input/output statement containing this specifier finishes execution, \texttt{iomsg\_variable} is defined as follows:

- If an error, end-of-file, or end-of-record condition occurs, the variable is assigned an explanatory message as if by assignment.
- If no such condition occurs, the value of the variable is unchanged.

\texttt{IOSTAT=} \texttt{ios}

is an input/output status specifier that specifies the status of the input/output operation. \texttt{ios} is an integer variable. When the input/output statement containing this specifier finishes execution, \texttt{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 \texttt{ios} defined for the \texttt{IOSTAT=} specifier of the asynchronous data transfer statement is not required to be identical to the \texttt{ios} defined for the \texttt{IOSTAT=} specifier of the matching \texttt{WAIT} statement.

\textbf{Rules}

The matching \texttt{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

- “Asynchronous Input/Output” on page 183
- Implementation details of XL Fortran Input/Output in the XL Fortran Optimization and Programming 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.

Syntax

```plaintext
where_construct_name: WHERE (mask_expr) WHERE_assignment_statement
```

Notes:

1 Fortran 95 (where_construct_name).

`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 103 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 105 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.

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:
! 

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

--- End of Fortran 95 ---

Related information
- “WHERE construct” on page 103
- “ELSEWHERE” on page 290
- “END (Construct)” on page 293, for details on the END WHERE statement

WRITE

Purpose
The WRITE statement is a data transfer output statement.

Syntax

```
WRITE((--io_control_list--) [output_item_list])
```

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 object 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 429

---

**Fortran 2003 Standard**

An output_item must not be a procedure pointer.

---

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.

---

**IBM Extension**

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**

Fortran 95 does not permit assigning of a statement label.

---

426  XL Fortran Language Reference
• 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.

POS=integer_expr

integer_expr is an integer expression greater than 0. POS= specifies the file position of the file storage unit to be written in a file connected for stream access. You must not use POS= for a file that cannot be positioned.

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. You must not specify REC= in data transfer statements that specify a unit connected for stream access, or use the POS= specifier.

ForTRAN 2003 Standard

IOMSG= iomsg_variable

is an input/output status specifier that specifies the message returned by the input/output operation. iomsg_variable is a scalar default character variable. It must not be a use-associated nonpointer protected variable. When the input/output statement containing this specifier finishes execution, iomsg_variable is defined as follows:

• If an error, end-of-file, or end-of-record condition occurs, the variable is assigned an explanatory message as if by assignment.

• If no such condition occurs, the value of the variable is unchanged.

End of ForTRAN 2003 Standard
IOSTAT= \texttt{ios}

is an input/output status specifier that specifies the status of the
input/output operation. \texttt{ios} is an integer variable. Coding the IOSTAT=
specifier suppresses error messages. When the statement finishes execution,
\texttt{ios} is defined with:
- A zero value if no error condition occurs
- A positive value if an error occurs.

---

**IBM Extension**

\texttt{ID= integer_variable}

indicates that the data transfer is to be done asynchronously. The
\texttt{integer_variable} is an integer variable. If no error is encountered, the
\texttt{integer_variable} is defined with a value after executing the asynchronous
data transfer statement. This value must be used in the matching \texttt{WAIT}
statement.

Asynchronous data transfer must either be direct unformatted, sequential
unformatted, or stream 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 \texttt{integer_variable} must not be
associated with any entity in the data transfer I/O list, or with a \texttt{do_variable}
of an \texttt{io_implied_do} in the data transfer I/O list. If the \texttt{integer_variable} is an
array element reference, its subscript values must not be affected by the
data transfer, the \texttt{io_implied_do} processing, or the definition or evaluation of
any other specifier in the \texttt{io_control_spec}.

---

**End of IBM Extension**

---

**Fortran 2003 Standard**

\texttt{DELIM= char_expr}

specifies what delimiter, if any, is used to delimit character constants
written with list-directed or namelist formatting. \texttt{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 \texttt{DELIM=}
specifier is permitted only for files being connected for formatted
input/output, although it is ignored during input of a formatted record.

---

**End of Fortran 2003 Standard**

---

\texttt{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 \texttt{ERR=} specifier suppresses error messages.

---

**IBM Extension**

\texttt{NUM= integer_variable}

is a number specifier that specifies the number of bytes of data transmitted
between the I/O list and the file. \texttt{integer_variable} is an integer variable. 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.

---------- End of IBM Extension ----------

[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 or formatted stream 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

\[
\rightarrow(\rightarrow do_{object\_list} \rightarrow, do_{variable} = arith_{expr1}, arith_{expr2} \rightarrow) \rightarrow
\]

\[
[,[,] arith_{expr3}]\]

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

IBM Extension

If a NUM= specifier is present, neither a format specifier nor a namelist specifier can be present.

End of IBM Extension

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.

IBM Extension

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.

End of IBM Extension

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

- “Asynchronous Input/Output” on page 183
- Implementation details of XL Fortran Input/Output in the XL Fortran Optimization and Programming Guide
- “Conditions and IOSTAT values” on page 186
- Chapter 8, “XL Fortran Input/Output,” on page 177
- “READ” on page 377
- “WAIT” on page 421
- Setting run-time options for Input/Output in the XL Fortran Compiler Reference
- “Deleted features” on page 746
Chapter 11. Directives

IBM Extension

This section provides an alphabetical reference to non-SMP directives that apply to all platforms. For a complete listing and description of SMP and thread-safe directives, see the detailed directive descriptions in the IBM Fortran Optimization and Programming Guide. For a detailed description of directives exclusive to the PowerPC® platform, see Chapter 12, “Hardware-specific directives,” on page 465.

Comment and noncomment form directives

XL Fortran directives belong to one of two groups: comment form directives and noncomment form directives.

Comment form directives

This section describes the format of comment form directives. The non-SMP comment form directives include the following:

<table>
<thead>
<tr>
<th>COLLABSE</th>
<th>SNAPSHOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOURCEFORM</td>
<td>SUBSCRIPTORDER</td>
</tr>
</tbody>
</table>

Additional comment form directives can be found in "Directives and optimization" on page 436.

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.

Rules

The default value for the trigger_constant is IBM*.

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 XL Fortran Compiler Reference for more details.

The compiler treats all comment form directives, with the exception of those using 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.
XL Fortran supports the OpenMP specification, as understood and interpreted by IBM. 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.

XL Fortran also includes the trigger_constants 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, xlf90_r, or xlf95_r invocation commands; we recommend its use with the THREADLOCAL directive.

XL Fortran 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 -directive compiler option will enable the trigger constant the vendor has selected. Refer to the directive compiler option in the XL Fortran Compiler Reference for details on specifying alternative trigger_constants.

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.

!IBM* 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:

!IBM* INDEPENDENT &
!TRIGGER&, REDUCTION (X) &
!IBM*&, NEW (I)

The above is equivalent to:

!IBM* INDEPENDENT, REDUCTION (X), NEW (I)

provided both IBM* and TRIGGER are active trigger_constants.

For more information, see “Lines and source formats” on page 5.

You can specify a directive as a free source form or fixed source form comment, depending on the current source form.

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 \textit{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 \textit{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 \textit{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 \textit{trigger_constant}.

The \textit{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 \textit{“Fixed source form” on page 6}.

\textbf{Free source form rules:} The \textit{trigger_head} is !. The maximum length of the \textit{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 \textit{directive_trigger} must appear at the beginning of all continuation lines. If you are beginning a continuation line with an ampersand, the \textit{directive_trigger} must precede the ampersand. For example:

\begin{verbatim}
!IBM* INDEPENDENT &
!IBM* & REDUCTION (X) &
!IBM* & NEW (I)
\end{verbatim}

For more information, see \textit{“Free source form” on page 9}.

\textbf{Noncomment form directives}

This section describes the format of noncomment form directives, which include the following:

\begin{verbatim}
EJECT INCLUDE
#LINE @PROCESS
\end{verbatim}

\textbf{Format}

\begin{verbatim}
-> directive <-
\end{verbatim}

\textbf{Rules}

The compiler always recognizes noncomment form directives.

Noncomment form directives 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.
Directives and optimization

The following are comment form directives useful for optimizing programs. See Optimizing XL compiler programs in the XL Fortran Optimization and Programming Guide and the compiler options that affect performance.

Assertive directives

Assertive directives gather information about source code that is otherwise unavailable to the compiler. Providing this information can increase performance.

<table>
<thead>
<tr>
<th>ASSERT</th>
<th>CNCALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDEPENDENT</td>
<td>PERMUTATION</td>
</tr>
</tbody>
</table>

Directives for Loop Optimization

The following directives provide different methods for loop optimization:

<table>
<thead>
<tr>
<th>BLOCK_LOOP</th>
<th>LOOPID</th>
</tr>
</thead>
<tbody>
<tr>
<td>STREAM_UNROLL</td>
<td>UNROLL</td>
</tr>
<tr>
<td>UNROLL_AND_FUSE</td>
<td>PREFETCH directives</td>
</tr>
</tbody>
</table>

Detailed directive descriptions

**ASSERT**

**Purpose**
The ASSERT directive provides the compiler with characteristics of DO loops that can assist in optimizing source code.

The ASSERT directive takes effect when you specify the -qhot or -qsmp compiler options.

**Syntax**

```
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

designates 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.
- If 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 accessing distinct areas of a stream access file; or
  - If the two I/O operations are accessing 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:

```fortran
!IBM* ASSERT (ITERCNT(10))
!IBM* INDEPENDENT, REDUCTION (A)
!IBM* 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:**

```fortran
! An example of the ASSERT directive with NODEPS.
PROGRAM EX1
   INTEGER A(100)
!IBM* ASSERT (NODEPS)
```
Example 2:

! An example of the ASSERT directive with NODEPS and ITERCNT.
SUBROUTINE SUB2 (N)
   INTEGER A(N)
!IBM* 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 [XL Fortran Compiler Reference](#)
- `qdirective` option in the [XL Fortran Compiler Reference](#)
- “Loop parallelization” on page 313

**BLOCK_LOOP**

**Purpose**

The BLOCK_LOOP directive allows you to exert greater control over optimizations on a specific DO loop inside a loop nest. Using a technique called blocking, the BLOCK_LOOP directive separates large iteration count DO loops into smaller iteration groups. Execution of these smaller groups can increase the efficiency of cache space use and augment performance.

Applying BLOCK_LOOP to a loop with dependencies, or a loop with alternate entry or exit points will produce unexpected results.

The BLOCK_LOOP directive takes effect only when the `-qhot`, `-qipa`, or `-qsmp` compiler option is specified.

**Syntax**

```
>>> BLOCK_LOOP ( \n    n \n  ) \n```

- **n** is a positive integer expression as the size of the iteration group.
- **name** a unique identifier in the same scoping unit as BLOCK_LOOP, that you can create using the LOOPID directive.

If you do not specify name, blocking occurs on the first DO loop immediately following the BLOCK_LOOP directive.
Rules
For loop blocking to occur, a BLOCK_LOOP directive must immediately precede a DO loop.

You must not specify the BLOCK_LOOP directive more than once.

You must not specify the BLOCK_LOOP directive for a DO WHILE loop or an infinite DO loop.

Examples
! Loop Tiling for Multi-level Memory Heirarchy

INTEGER :: M, N, i, j, k
M = 1000
N = 1000

!IBM* BLOCK_LOOP(L3_cache_size, L3_cache_block)
do i = 1, N
!IBM* LOOPID(L3_cache_block)
!IBM* BLOCK_LOOP(L2_cache_size, L2_cache_block)
do j = 1, N
!IBM* LOOPID(L2_cache_block)
do k = 1, M
do l = 1, M
end do
end do
end do
end do
end

! The compiler generated code would be equivalent to:
do index1 = 1, M, L3_cache_size
do i = 1, N
do index2 = index1, min(index1 + L3_cache_size, M), L2_cache_size
do j = 1, N
do k = index2, min(index2 + L2_cache_size, M)
do l = 1, M
end do
end do
end do
end do
end do

Related information
- For additional methods of optimizing loops, see the STREAM UNROLL, the UNROLL and UNROLL_AND_FUSE directives.
**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.

**Syntax**

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>N</td>
<td>C</td>
<td>A</td>
<td>L</td>
<td>L</td>
<td></td>
</tr>
</tbody>
</table>

```

**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 the [ASSERT](#) directive 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.

```fortran
PROGRAM EX3
  INT  A(100)
!IBM*  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 444
- -qdirective in the [XL Fortran Compiler Reference](#)
- -qhot Option in the [XL Fortran Compiler Reference](#)
- -qsmp Option in the [XL Fortran Compiler Reference](#)
- “DO” on page 280
- “Loop parallelization” on page 313
COLLAPSE

Purpose
The COLLAPSE directive reduces an entire array dimension to a single element by specifying that only the element in the lower bound of an array dimension is accessible. If you do not specify a lower bound, the default lower bound is one.

Used with discretion, the COLLAPSE directive can facilitate an increase in performance by reducing repetitive memory access associated with multiple-dimension arrays.

Syntax

\[
\texttt{COLLAPSE} (\texttt{collapse\_array\_list})
\]

where \texttt{collapse\_array} is:

\[
\texttt{array\_name} (\texttt{expression\_list})
\]

where \texttt{expression\_list} is a comma separated list of \texttt{expression}.

\texttt{array\_name}

is the array name.

\texttt{expression}

is a constant scalar integer expression. You may only specify positive integer values.

Rules
The COLLAPSE directive must contain at least one array.

The COLLAPSE directive applies only to the scoping unit in which it is specified. The declarations of arrays contained in a COLLAPSE directive must appear in the same scoping unit as the directive. An array that is accessible in a scoping unit by use or host association must not specified in a COLLAPSE directive in that scoping unit.

The lowest value you can specify in \texttt{expression\_list} is one. The highest value must not be greater than the number of dimensions in the corresponding array.

A single scoping unit can contain multiple COLLAPSE declarations, though you can only specify an array once for a particular scoping unit.

You can not specify an array in both a COLLAPSE directive and an EQUIVALENCE statement.

You can not use the COLLAPSE directive with arrays that are components of derived types.

If you apply both the COLLAPSE and SUBSCRIPTORDER directives to an array, you must specify the SUBSCRIPTORDER directive first.
The **COLLAPSE** directive applies to:

- Assumed-shape arrays in which all lower bounds must be constant expressions.
- Explicit-shape arrays in which all lower bounds must be constant expressions.

**Examples**

**Example 1:** In the following example, the **COLLAPSE** directive is applied to the explicit-shape arrays `A` and `B`. Referencing `A(m,2:100,2:100)` and `B(m,2:100,2:100)` in the inner loops, become `A(m,1,1)` and `B(m,1,1).

```fortran
!IBM* COLLAPSE(A(2,3),B(2,3))
REAL*8 A(5,100,100), B(5,100,100), c(5,100,100)
DO I=1,100
  DO J=1,100
    DO M=1,5
      A(M,J,I) = SIN(C(M,J,I))
      B(M,J,I) = COS(C(M,J,I))
    END DO
  END DO
END DO
DO M=1,5
  DO N=1,M
  END DO
END DO
END DO
END DO
END DO
```

**Related information**

For more information on the **SUBSCRIPTORDER** directive, see "SUBSCRIPTORDER" on page 458.

---

**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.

**Syntax**

```
   EJECT
```

**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.

---

**INCLUDE**

**Purpose**

The **INCLUDE** compiler directive inserts a specified statement or a group of statements into a program unit.
Syntax

```
#include <char_literal_constant> (name) n
```

name, char_literal_constant (delimiters are optional)
specifies filename, the name of an include file

You are not required to specify the full path of the desired file, but must
specify the file extension if one exists.

name must contain only characters allowable in the XL Fortran character
set. See “Characters” on page 3 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.

Conditional include allows you to selectively activate INCLUDE directives within
Fortran source during compilation. Specify the files to include using 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 file system locates the specified 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 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 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 abcd
INCLUDE './Abc' ! includes file Abc from parent directory

Related information
[qci Option in the XL Fortran Compiler Reference]

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.

Type
This directive only takes effect if you specify either the -qsmp or -qhot compiler option.

Syntax

| INDEPENDENT | NEW (named_variable_list) | REDUCTION (named_variable_list) |

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.

• If 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 accessing distinct areas of a stream access file; or
  – If the two I/O operations are accessing 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:

\[
\begin{align*}
\text{reduction_var_ref} & = \text{expr} \\
\text{reduction_var_ref} & = \text{reduction_var_ref} \text{ reduction_op } \text{expr} \\
\text{reduction_var_ref} & = \text{reduction_function}(\text{expr}, \text{reduction_var_ref}) \\
\text{reduction_var_ref} & = \text{reduction_function}(\text{reduction_var_ref}, \text{expr})
\end{align*}
\]

where:
- \text{reduction_var_ref} is a variable or subobject of a variable that appears in a REDUCTION clause
- \text{reduction_op} is one of: +, −, *, .AND., .OR., .EQV., .NEQV., or .XOR.
- \text{reduction_function} is one of: MAX, MIN, IAND, IOR, or IEX

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 \text{reduction_var_refs} 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
FORALL (I=1:9:2) A(I)=A(I+1) ! The NEW clause cannot be
!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 following the INDEPENDENT must
   B(M,N)=M+N+N ! be either a DO or FORALL
END DO
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)
!IBM* INDEPENDENT, REDUCTION(A), NEW(J) ! Example showing an array used for a reduction variable
DO I=1,100
   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 119
- “FORALL” on page 311
- `-directive` in the [XL Fortran Compiler Reference](https://example.com)
- `-qhot Option` in the [XL Fortran Compiler Reference](https://example.com)
- `-qsmp Option` in the [XL Fortran Compiler Reference](https://example.com)

**#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.
Syntax

The #line directive is a noncomment directive and follows the syntax rules for this type of directive.

\( \text{line_number} \)

is a positive, unsigned integer literal constant without a KIND parameter. You must specify line_number.

\( \text{filename} \)

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.

\( \text{line_number} \)

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"
...```
The file test.h contains:
! File test.h line 1
PRINT*,1 ! Syntax Error
PRINT*,2

After the C preprocessor 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
PRINT*,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
PRINT*,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:
1 2 |PRINT*,1
!Syntax error

......a................
a - "test.h", line 2.6: 1515-019 (S) Syntax is incorrect.

Related information
- `-d` Option in the [XL Fortran Compiler Reference]
- [Passing Fortran Files through the C Preprocessor] in the [XL Fortran Compiler Reference]

**LOOPID**

**Purpose**
The LOOPID directive allows you to assign a unique identifier to loop within a scoping unit. You can use the identifier to direct loop transformations. The `-qreport` compiler option can use the identifier you create to provide reports on loop transformations.

**Syntax**

```
LOOPID(---name---)
```
name is an identifier that must be unique within the scoping unit.

**Rules**
The LOOPID directive must immediately precede a **BLOCK_LOOP** directive or **DO** construct.

You must not specify a LOOPID directive more than once for a given loop.

You must not specify a LOOPID directive for DO constructs without control statements, **DO WHILE** constructs, or an infinite DO.

**Related information**
- For additional methods of optimizing loops, see the **BLOCK_LOOP** **STREAM** **UNROLL**, **UNROLL** and the **UNROLL_AND_FUSE** directives.

**NEW**

--- IBM Extension ---

**Purpose**
Use the NEW directive to specify which variables should be local in a PARALLEL DO loop or a PARALLEL SECTIONS construct. This directive performs the same function as the PRIVATE clause of the PARALLEL DO directive and PARALLEL SECTIONS directive.

**Class**
The NEW directive only takes effect if you specify the -qsmp compiler option.

**Syntax**

```
NEW named_variable_list
```

**Rules**
The NEW directive must immediately follow either a PARALLEL DO directive or a PARALLEL SECTIONS directive.

If you specify the NEW directive, you must specify the corresponding PARALLEL DO or PARALLEL SECTIONS directive with no clauses.

If the NEW directive follows the PARALLEL DO directive, the first noncomment line (not including other directives) following the NEW directive must be a DO loop. This line cannot be an infinite DO or DO WHILE loop.

A variable name in the named_variable_list of the NEW directive has the same restrictions as a variable name appearing in the PRIVATE clause of the PARALLEL DO directive or a PRIVATE clause of the PARALLEL SECTIONS directive. See the sections on the PARALLEL DO directive and the PARALLEL SECTIONS construct in the **XL Fortran Optimization and Programming Guide**

**Examples**

```fortran
INTEGER A(10), C(10)
REAL B(10)
INTEGER FUNC(100)
!SMP$ PARALLEL DO
!SMP$ NEW I, TMP
```
DO I = 1, 10
   TMP = A(I) + COS(B(I))
   C(I) = TMP + FUNC(I)
END DO

NOSIMD

Purpose
The NOSIMD directive prohibits the compiler from automatically generating Vector Multimedia eXtension (VMX) instructions in the loop immediately following the directive, or in the FORALL construct.

Syntax

```
>>> NOSIMD <<<
```

Rules
The first noncomment line (not including other directives) following the NOSIMD 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 NOSIMD directive applies only to the DO loop that is immediately following the directive and does not apply to any nested DO loops.

You can use the NOSIMD directive together with loop optimization and SMP directives.

Examples

```fortran
SUBROUTINE VEC (A, B)
   REAL*8 A(200), B(200)
   !IBM* NOSIMD
   DO (N = 1:200), B(N) = B(N) / A(N)
END SUBROUTINE
```

Related information
Please refer to the `-qhot=simd` compiler option for information on controlling VMX support for an entire application.

NOVECTOR

Purpose
The NOVECTOR directive prohibits the compiler from auto-vectorizing the loop immediately following the directive. Auto-vectorization involves converting certain operations performed in a loop and on successive array elements into a call to a routine that computes several results simultaneously.
Syntax

```fortran
!IBM* NOVECTOR
```

Rules

The first noncomment line (not including other directives) following the `NOVECTOR` 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 `NOVECTOR` directive applies only to the `DO` loop that is immediately following the directive and does not apply to any nested `DO` loops, or nested `FORALL` construct or statement.

You can use the `NOVECTOR` directive together with loop optimization and SMP directives.

Examples

```fortran
SUBROUTINE VEC (A, B)
    REAL*8 A(200), B(200)
    !IBM* NOVECTOR
    DO N = 1, 200
        B(N) = B(N) / A(N)
    END DO
END SUBROUTINE
```

Related information

Please refer to the `-qhot=vector` compiler option for information on controlling auto-vectorization for an entire application.

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.

Syntax

```fortran
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

```fortran
PROGRAM EX3
    INTEGER A(100), B(100)
    !IBM* 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 XL Fortran Compiler Reference](#)
- [qsmp Option in the XL Fortran Compiler Reference](#)
- “DO” on page 280

@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.

**Syntax**

```
@PROCESS option [suboption_list]
```

- `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.
SNAPSHOT

Purpose
You can use the SNAPSHOT directive to specify a safe location where a breakpoint can be set with a debug program, and provide a set of variables that must remain visible to the debug program. The SNAPSHOT directive provides support for the -qsmp compiler option, though you can use it in a non-multi-threaded program.

There may be a slight performance hit at the point where the SNAPSHOT directive is set, because the variables must be kept in memory for the debug program to access. Variables made visible by the SNAPSHOT directive are read-only. Undefined behavior will occur if these variables are modified through the debugger. Use with discretion.

At high optimization levels, the SNAPSHOT directive does not consistently preserve the contents of variables with a static storage class.

Syntax

```fortran
SNAPSHOT(named_variable_list)
```

`named_variable`

is a named variable that must be accessible in the current scope.

Rules
To use the SNAPSHOT directive, you must specify the -qdbg compiler option at compilation.

Examples
Example 1: In the following example, the SNAPSHOT directive is used to monitor the value of private variables.

```fortran
INTEGER :: IDX
INTEGER :: OMP_GET_NUM_THREADS, OMP_GET_THREAD_NUM
INTEGER, ALLOCATABLE :: ARR(:)
!
!$OMP PARALLEL, PRIVATE(IDX)
!$OMP MASTER
ALLOCATE(ARR(OMP_GET_NUM_THREADS()))
!$OMP END MASTER
!$OMP BARRIER

IDX = OMP_GET_THREAD_NUM() + 1

!IBM* SNAPSHOT(IDX)!

ARR(IDX) = 2*IDX + 1

!$OMP END PARALLEL
```

Related information
See Compiler Option Details in the XL Fortran Compiler Reference for details on compiler options.
Example 2: In the following example, the **SNAPSHOT** directive is used to monitor the intermediate values in debugging the program.

```fortran
SUBROUTINE SHUFFLE(NTH, XDAT)
INTEGER, INTENT(IN) :: NTH
REAL, INTENT(INOUT) :: XDAT(:)
INTEGER :: I_TH, IDX, PART(1), I, J, LB, UB
INTEGER :: OMP_GET_THREAD_NUM
INTEGER(8) :: Y=1
REAL :: TEMP
!
CALL OMP_SET_NUM_THREADS(NTH)
PART = UBOUND(XDAT)/NTH
!
$OMP PARALLEL, PRIVATE(NUM_TH, I, J, LB, UB, IDX, TEMP), SHARED(XDAT)
NUM_TH = OMP_GET_THREAD_NUM() + 1
LB = (NUM_TH - 1)*PART(1) + 1
UB = NUM_TH*PART(1)
!
DO I=LB, UB
!
$OMP CRITICAL
Y = MOD(65539_8*y, 2_8**31)
IDX = INT(REAL(Y)/REAL(2_8**31)*(UB - LB) + LB)
!
SNAPSHOT(i, y, idx, num_th, lb, ub)
!
$OMP END CRITICAL

TEMP = XDAT(I)
XDAT(I) = XDAT(IDX)
XDAT(IDX) = TEMP
ENDDO
!
SNAPSHOT(TEMP) ! The user can examine the value
! of the TEMP variable
!
$OMP END PARALLEL
END
```

**Related information**
See the [XL Fortran Compiler Reference](#) for details on the `-qdbg` compiler option.

**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.

**Syntax**

```
SOURCEFORM(source)
```

*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:
   ```fortran
   !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**
```fortran
@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:
```fortran
!CONVERT* SOURCEFORM(FREE(F90))
RESULT = A + B
```

**STREAM_UNROLL**

**Purpose**
The **STREAM_UNROLL** directive instructs the compiler to apply the combined functionality of software prefetch and loop unrolling to [**DO loops**] with a large iteration count. Stream unrolling functionality is available only on [POWER4™] platforms or higher, and optimizes **DO** loops to use multiple streams. You can specify the **STREAM_UNROLL** directive for both inner and outer **DO** loops, and the compiler will use an optimal number of streams to perform stream unrolling where applicable. Applying the **STREAM_UNROLL** directive to a loop with dependencies will produce unexpected results.
**Syntax**

```
STREAM_UNROLL(unroll_factor)
```

*unroll_factor*

The *unroll_factor* must be a positive scalar integer initialization expression. An *unroll_factor* of 1 disables loop unrolling. If you do not specify an *unroll_factor*, stream unrolling is compiler determined.

**Rules**

You must specify one of the following compiler options to enable loop unrolling:

- `-O3` or higher optimization level
- `-qhot`
- `-qsmp`

Note that if the `-qstrict` option is in effect, no stream unrolling will occur. If you want to enable stream unrolling with the `-qhot` option alone, you must also specify `-qnostrict`.

The `STREAM_UNROLL` directive must immediately precede a `DO` loop.

You must not specify the `STREAM_UNROLL` directive more than once, or combine the directive with `UNROLL`, `NOUNROLL`, `UNROLL_AND_FUSE`, or `NOUNROLL_AND_FUSE` directives for the same `DO` construct.

You must not specify the `STREAM_UNROLL` directive for a `DO WHILE` loop or an infinite `DO` loop.

**Examples**

The following is an example of how `STREAM_UNROLL` can increase performance.

```fortran
integer, dimension(1000) :: a, b, c
integer i, m, n

!IBM stream_unroll(4)
do i = 1, n
   a(i) = b(i) + c(i)
endo
done
```

An *unroll factor* reduces the number of iterations from `n` to `n/4`, as follows:

```fortran
m = n/4
do i = 1, n/4
   a(i) = b(i) + c(i)
   a(i+m) = b(i+m) + c(i+m)
   a(i+2*m) = b(i+2*m) + c(i+2*m)
   a(i+3*m) = b(i+3*m) + c(i+3*m)
endo
```

The increased number of read and store operations are distributed among a number of streams determined by the compiler, reducing computation time and boosting performance.
Related information
- For further information on using prefetch techniques in XL Fortran see the PRECEPT directive set.
- For additional methods on optimizing loops, see the BLOCK_LOOP, LOOPID, UNROLL and the UNROLL_AND_FUSE directives.

SUBSCRIPTORDER

Purpose
The SUBSCRIPTORDER directive rearranges the subscripts of an array. This results in a new array shape, since the directive changes the order of array dimensions in the declaration. All references to the array are correspondingly rearranged to match the new array shape.

Used with discretion, the SUBSCRIPTORDER directive may improve performance by increasing the number of cache hits and the amount of data prefetching. You may have to experiment with this directive until you find the arrangement that yields the most performance benefits. You may find SUBSCRIPTORDER especially useful when porting code originally intended for a non-cached hardware architecture.

In a cached hardware architecture, such as the PowerPC, an entire cache line of data is often loaded into the processor in order to access each data element. Changing the storage arrangement can be used to ensure that consecutively accessed elements are stored adjacently. This may result in a performance improvement, as there are more element accesses for each cache line referenced. Additionally, adjacently storing array elements which are consecutively accessed may help to better exploit the processor’s prefetching facility.

Syntax

```
SUBSCRIPTORDER(array_name(subscriptorder_number))
```

where `subscriptorder_array` is:

```
array_name(subscriptorder_number)
```

array name is the name of an array.

subscriptorder_number is an integer constant.

Rules
The SUBSCRIPTORDER directive must appear in a scoping unit preceding all declarations and references to the arrays in the subscriptorder_array list. The directive only applies to that scoping unit and must contain at least one array. If
multiple scoping units share an array, then you must apply the
SUBSCRIPTORDER directive to each of the applicable scoping units with
identical subscript arrangements. Examples of methods of array sharing between
scoping units include COMMON statements, USE statements, and subroutine
arguments.

The lowest subscript number in a subscriptorder_number list must be 1. The highest
number must be equal to the number of dimensions in the corresponding array.
Every integer number between these two limits, including the limits, signifies a
subscript number prior to rearrangement and must be included exactly once in the
list.

You must not apply a SUBSCRIPTORDER directive multiple times to a particular
array in a scoping unit.

You must maintain array shape conformance in passing arrays as actual arguments
to elemental procedures, if one of the arrays appears in a SUBSCRIPTORDER
directive. You must also adjust the actual arguments of the SHAPE, SIZE,
LBOUND, and UBOUND inquiry intrinsic procedures and of most
transformational intrinsic procedures.

You must manually modify data in input data files and in explicit initializations for
arrays that appear in the SUBSCRIPTORDER directive.

On arrays to which the COLLAPSE directive is also applied, the COLLAPSE
directive always refers to the pre-subscriptorder dimension numbers.

You must not rearrange the last dimension of an assumed-size array.

Examples
Example 1: In the following example, the SUBSCRIPTORDER directive is applied
to an explicit-shape array and swaps the subscripts in every reference to the array,
without affecting the program output.

```
!IBM* SUBSCRIPTORDER(A(2,1))
   INTEGER COUNT/1/, A(3,2)
   DO J = 1, 3
      DO K = 1, 2
         ! Inefficient coding: innermost index is accessing rightmost
         ! dimension. The subscriptorder directive compensates by
         ! swapping the subscripts in the array's declaration and
         ! access statements.
         !
         ! A(J,K) = COUNT
         PRINT*, J, K, A(J,K)
      END DO
      COUNT = COUNT + 1
   END DO
```

Without the directive above, the array shape is (3,2) and the array elements would
be stored in the following order:

```
A(1,1) A(2,1) A(3,1) A(1,2) A(2,2) A(3,2)
```

With the directive, the array shape is (2,3) and the array elements are stored in the
following order:

```
A(1,1) A(2,1) A(1,2) A(2,2) A(1,3) A(2,3)
```
Related information
For more information on the COLLAPSE directive, see "COLLAPSE" on page 441

UNROLL

Purpose
The UNROLL directive instructs the compiler to attempt loop unrolling where applicable. Loop unrolling replicates the body of the DO loop to reduce the number of iterations required to complete the loop.

You can control loop unrolling for an entire file using the -qunroll compiler option. Specifying the directive for a particular DO loop always overrides the compiler option.

Syntax

```
UNROLL

NOUNROLL

(unroll_factor)
```

unroll_factor
The unroll_factor must be a positive scalar integer initialization expression. An unroll_factor of 1 disables loop unrolling. If you do not specify an unroll_factor, loop unrolling is compiler determined.

Rules
You must specify one of the following compiler options to enable loop unrolling:
• -O3 or higher optimization level
• -qhot
• -qsmp

Note that if the -qstrict option is in effect, no loop unrolling will occur. If you want to enable loop unrolling with the -qhot option alone, you must also specify -qnostrict.

The UNROLL directive must immediately precede a DO loop.

You must not specify the UNROLL directive more than once, or combine the directive with NOUNROLL, STREAM_UNROLL, UNROLL_AND_FUSE or NOUNROLL_AND_FUSE directives for the same DO construct.

You must not specify the UNROLL directive for a DO WHILE loop or an infinite 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.

```fortran
!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:

```plaintext
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.

```plaintext
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**
- For additional methods of optimizing loops, see the `BLOCK_LOOP` and `LOOPID` directives.
- `STREAM UNROLL` and the `UNROLL_AND_FUSE` directives.

### UNROLL_AND_FUSE

**Purpose**
The `UNROLL_AND_FUSE` directive instructs the compiler to attempt a loop unroll and fuse where applicable. Loop unrolling replicates the body of multiple DO loops and combines the necessary iterations into a single unrolled loop. Using a fused loop can minimize the required number of loop iterations, while reducing the frequency of cache misses. Applying the `UNROLL_AND_FUSE` directive to a loop with dependencies will produce unexpected results.

**Syntax**

```
UNROLL_AND_FUSE

NOUNROLL_AND_FUSE
```

`unroll_factor`
The `unroll_factor` must be a positive scalar integer initialization expression. An `unroll_factor` of 1 disables loop unrolling. If you do not specify an `unroll_factor`, loop unrolling is compiler determined.
Rules
You must specify one of the following compiler options to enable loop unrolling:
- -O3 or higher optimization level
- -qhot
- -qsmp

Note that if the -qstrict option is in effect, no loop unrolling will occur. If you want to enable loop unrolling with the -qhot option alone, you must also specify -qnostrict.

The UNROLL_AND_FUSE directive must immediately precede a DO loop.

You must not specify the UNROLL_AND_FUSE directive for the innermost DO loop.

You must not specify the UNROLL_AND_FUSE directive more than once, or combine the directive with NOUNROLL_AND_FUSE, NOUNROLL, UNROLL or STREAM_UNROLL directives for the same DO construct.

You must not specify the UNROLL_AND_FUSE directive for a DO WHILE loop or an infinite DO loop.

Examples
Example 1: In the following example, the UNROLL_AND_FUSE directive replicates and fuses the body of the loop. This reduces the number of cache misses for Array B.

```
INTEGER, DIMENSION(1000, 1000) :: A, B, C
!IBM* UNROLL_AND_FUSE(2)
DO I = 1, 1000
  DO J = 1, 1000
    A(J,I) = B(I,J) * C(J,I)
  END DO
END DO
END DO
END
```

The DO loop below shows a possible result of applying the UNROLL_AND_FUSE directive.

```
DO I = 1, 1000, 2
  DO J = 1, 1000
    A(J,I) = B(I,J) * C(J,I)
    A(J,I+1) = B(I+1,J) * C(J,I+1)
  END DO
END DO
END
```

Example 2: The following example uses multiple UNROLL_AND_FUSE directives:

```
INTEGER, DIMENSION(1000, 1000) :: A, B, C, D, H
!IBM* UNROLL_AND_FUSE(4)
DO I = 1, 1000
!IBM* UNROLL_AND_FUSE(2)
  DO J = 1, 1000
    DO K = 1, 1000
      A(J,I) = B(I,J) * C(J,I) + D(J,K) * H(I,K)
    END DO
  END DO
END DO
END
```

Related information
- For additional methods of optimizing loops, see the BLOCK_LOOP, LOOPID, STREAM_UNROLL and the UNROLL directives.
Chapter 12. Hardware-specific directives

This section provides an alphabetical reference to hardware-specific compiler directives. Unless otherwise noted, a directive will function on any supported hardware. This section contains the following directives:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>“CACHE_ZERO”</td>
<td>“LIGHT_SYNC” on page 466</td>
</tr>
<tr>
<td>“ISYNC” on page 466</td>
<td>“PREFETCH” on page 466</td>
</tr>
<tr>
<td>“PROTECTED STREAM” on page 469</td>
<td></td>
</tr>
</tbody>
</table>

CACHE_ZERO

**Purpose**

The CACHE_ZERO directive invokes the machine instruction, data cache block set to zero (dcbz). This instruction sets the data cache block corresponding to the variables you specify to zero. Use this directive with discretion.

**Syntax**

```
----CACHE_ZERO(---cv_var_list---)
```

*cv_var* is a variable associated with the cache block that is set to zero. The variable must be a data object with a determinable storage address. The variable cannot be a procedure name, subroutine name, module name, function name, constant, label, zero-sized string, or an array with vector subscripts.

**Examples**

In the following example, assume that array *ARRA* has already been loaded into a cache block that you want to set to zero. The data in the cache block is then set to zero.

```plaintext
real(4) :: arrA(2**5)
!
!IBM* CACHE_ZERO(arrA(1)) ! set data in cache block to zero
```

EIEIO

**Purpose**

Enforce In-order Execution of Input/Output (EIEIO).

The EIEIO directive allows you to specify that all I/O storage access instructions preceding the directive complete before any I/O access instruction subsequent to the directive can begin. Use EIEIO when managing shared data instruction where the execution order of load/store access is significant.

EIEIO can provide the necessary functionality for controlling I/O stores without the cost to performance that can occur with other synchronization instructions.
Syntax

```
EIEIO
```

**ISYNC**

**Purpose**
The **ISYNC** directive enables you to discard any prefetched instructions after all preceding instructions complete. Subsequent instructions are fetched or refetched from storage and execute in the context of previous instructions. The directive only affects the processor that executes **ISYNC**.

```
ISYNC
```

**LIGHT_SYNC**

**Purpose**
The **LIGHT_SYNC** directive ensures that all stores prior to **LIGHT_SYNC** complete before any new instructions can be executed on the processor that executed the **LIGHT_SYNC** directive. This allows you to synchronize between multiple processors with minimal performance impact, as **LIGHT_SYNC** does not wait for confirmation from each processor.

```
LIGHT_SYNC
```

**PREFETCH**

**Purpose**
You can use prefetching to instruct the compiler to load specific data from main memory into the cache before the data is referenced. Some prefetching can be done automatically by hardware that is POWER3™ and above, but since compiler-assisted software prefetching can use information directly from your source code, specifying the directive can significantly reduce the number of cache misses.

XL Fortran provides the following directives for compiler-assisted software prefetching:
• 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 or higher machine, `PREFETCH_BY_LOAD` enables hardware-assisted prefetching.

• The `PREFETCH_BY_STREAM` prefetch technique uses the POWER4 prefetch engine to recognize sequential access to adjacent cache lines and then requests anticipated lines from deeper levels of the memory hierarchy. This technique establishes a path or stream as repeated references to main memory are made, increasing the depth of the prefetch until enough lines are loaded into the cache. To fetch data from decremental memory addresses, use the `PREFETCH_BY_STREAM_BACKWARD` directive. To fetch data from incremental memory addresses, use the `PREFETCH_BY_STREAM_FORWARD` directive. The use of this streamed prefetch to load data from main memory into the cache can reduce or eliminate load latency.

• The `PREFETCH_FOR_LOAD` directive prefetches data into the cache for reading by way of a cache prefetch instruction.

• The `PREFETCH_FOR_STORE` directive prefetches data into the cache for writing by way of a cache prefetch instruction.

Syntax

The `PREFETCH` directive can take the following forms:

```
++PREFETCH_BY_LOAD—(—prefetch_variable_list—)++
```

```
++PREFETCH_FOR_LOAD—(—prefetch_variable_list—)++
```

```
++PREFETCH_FOR_STORE—(—prefetch_variable_list—)++
```

```
++PREFETCH_BY_STREAM_BACKWARD—(—prefetch_variable—)++
```

```
++PREFETCH_BY_STREAM_FORWARD—(—prefetch_variable—)++
```

`prefetch_variable` is a variable to be prefetched. The variable must be a data object with a determinable storage address. The variable can be of any data type, including intrinsic and derived data types. The variable cannot be a procedure name, subroutine name, module name, function name, constant, label, zero-sized string, or an array with a vector subscript.
Rules

To use the `PREFETCH_BY_STREAM_BACKWARD`, `PREFETCH_BY_STREAM_FORWARD`, `PREFETCH_FOR_LOAD` and `PREFETCH_FOR_STORE` directives, you must compile for PowerPC hardware.

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.

To maximize the effectiveness of the prefetch directives, it is recommended that you specify the `LIGHT_SYNC` directive after a single prefetch or at the end of a series of prefetches.

Examples

**Example 1:** 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)`.)

```fortran
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
```

468  XL Fortran Language Reference
DO I = 1, 2 ** 5

! 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

Example 2: In this example, assume that the total cache line’s size is 256 bytes, and that none of the declared data items are initially stored in the cache or register. All elements of array ARRA and ARRC will then be read into the cache.

PROGRAM PREFETCH_STREAM
REAL*4 A, B, C, TEMP
REAL*4 ARRA(2**5), ARRC(2**5), ARRB(2**10)
INTEGER*4 I, K

! All elements of ARRA and ARRC are read into the cache.
!IBM* PREFETCH_BY_STREAM_FORWARD(ARRA(1))
! You can substitute PREFETCH_BY_STREAM_BACKWARD (ARRC(2**5)) to read all
! elements of ARRA and ARRC into the cache.
K = 32
DO I = 1, 2**5
    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 PREFETCH_STREAM

Related information
For information on applying prefetch techniques to loops with a large iteration count, see the STREAM_UNROLL directive.

PROTECTED STREAM

Purpose
The PROTECTED STREAM directives allow you to manage protected streams. These streams are protected from being replaced by any hardware-detected streams.

XL Fortran provides the following protected stream directives:

- The PROTECTED_UNLIMITED_STREAM_SET_GO_FORWARD directive establishes an unlimited-length protected stream that begins with the cache line at the specified prefetch variable and fetches from increasing memory addresses. The PROTECTED_UNLIMITED_STREAM_SET_GO_BACKWARD directive fetches from decreasing memory addresses.
- The PROTECTED_STREAM_SET_FORWARD directive establishes a limited-length protected stream that begins with the cache line at the specified prefetch variable and fetches from increasing memory addresses. The PROTECTED_STREAM_SET_BACKWARD directive fetches from decreasing memory addresses.
• The PROTECTED_STREAM_COUNT directive sets the number of cache lines for the specified limited-length stream.
• The PROTECTED_STREAM_GO directive starts to prefetch all limited-length streams.
• The PROTECTED_STREAM_STOP directive stops prefetching the specified protected stream.
• The PROTECTED_STREAM_STOP_ALL directive stops prefetching all protected streams.

Syntax

The PROTECTED directive can take the following forms:

```fortran
>>>PROTECTED_UNLIMITED_STREAM_SET_GO_FORWARD—(prefetch_variable,—stream_ID—)<<<
```

Note: Valid for PowerPC 970 and POWER5™.

```fortran
>>>PROTECTED_UNLIMITED_STREAM_SET_GO_BACKWARD—(prefetch_variable,—stream_ID—)<<<
```

Note: Valid for PowerPC 970 and POWER5.

```fortran
>>>PROTECTED_STREAM_SET_FORWARD—(prefetch_variable,—stream_ID—)<<<
```

Note: Valid for POWER5 only.

```fortran
>>>PROTECTED_STREAM_SET_BACKWARD—(prefetch_variable,—stream_ID—)<<<
```

Note: Valid for POWER5 only.

```fortran
>>>PROTECTED_STREAM_COUNT—(unit_count,—stream_ID—)<<<
```

Note: Valid for POWER5 only.

```fortran
>>>PROTECTED_STREAM_GO—
```

Note: Valid for POWER5 only.
Note: Valid for POWER5 only.

```
>>> PROTECTED_STREAM_STOP(-stream_ID-_)
```

Note: Valid for POWER5 only.

```
>>> PROTECTED_STREAM_STOP_ALL
```

Note: Valid for POWER5 only.

`prefetch_variable` is a variable to be prefetched. The variable must be a data object with a determinable storage address. The variable can be of any data type, including intrinsic and derived data types. The variable cannot be a procedure name, subroutine name, module name, function name, literal constant, label, zero-sized string, zero-length array, or array with a vector subscript.

`stream_ID` is the ID for the prefetched stream. It must be scalar and of type integer. It can be any number from 0 to 7.

`unit_count` is the number of cache lines for the limited-length protected stream. It must be scalar and of type integer. It can be any number from 0 to 1023. For a stream that is larger than 1024 cache lines, use the `PROTECTED_UNLIMITED_STREAM` directives instead of the `PROTECTED_STREAM` directives.

Related information

For information on applying prefetch techniques to loops with a large iteration count, see the `STREAM_UNROLL` directive.
Chapter 13. Intrinsic procedures

Fortran defines a number of procedures, called intrinsic procedures, that are available to any program. This section provides an alphabetical reference to these procedures.

Related information:
1. “Intrinsic procedures” on page 156 provides background information that you may need to be familiar with before proceeding with this section.
2. “INTRINSIC” on page 344 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
- ASSOCIATED
- BIT_SIZE
- COMMAND_ARGUMENT_COUNT
- DIGITS
- EPSILON
- HUGE
- KIND
- LBOUND
- LEN
- LOC
- MAXEXponent
- MINEXponent
- NEW_LINE
- NUM_PARTHDS
- NUM_USRTHDS
- PRECISION
- PRESENT
- RADIX
- RANGE
- SHAPE
- SIZE
- SIZEOF
- TINY
- UBOUND

Notes:
1. IBM Extension.

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.
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.

Notes:
1. IBM Extension.

**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.
Intrinsic subroutines

Some intrinsic procedures are subroutines. They perform a variety of tasks.

Notes:
1. IBM Extension.
2. Fortran 95.

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 \( k \)
IBM Extension

XL Fortran implements the following \( s \) parameters for the XL Fortran integer kind type parameters:

<table>
<thead>
<tr>
<th>Integer Kind Parameter</th>
<th>( s ) Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
</tr>
</tbody>
</table>

The following intrinsic functions use this model:

- BTEST
- IAND
- IBCLR
- IBITS
- BTEST
- IBSET
- ISHFT
- IEOR
- IOR
- ISHFTC
- IBSET
- IOR
- MVBITS
- NOT

**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 (\( \pm 1 \))
- \( q \) is the number of digits (positive integer)
- \( w_k \) is a nonnegative digit < \( r \)
- \( r \) is the radix

IBM Extension

XL Fortran implements this model with the following \( r \) and \( q \) parameters:

<table>
<thead>
<tr>
<th>Integer Kind Parameter</th>
<th>( r ) Parameter</th>
<th>( q ) Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>31</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>63</td>
</tr>
</tbody>
</table>

The following intrinsic functions use this model:

- DIGITS
- RADIX
- RANGE

---

476 XL Fortran Language Reference
Real data model

\[
x = \begin{cases} 
0 & \text{or} \\
 s \times b^e \sum_{k=1}^{p} f_k b^{-k} & 
\end{cases}
\]

- \(x\) is the real value
- \(s\) is the sign (±1)
- \(b\) is an integer > 1
- \(e\) is an integer, where \(e_{\min} \leq e \leq e_{\max}\)
- \(p\) is an integer > 1
- \(f_k\) is a nonnegative integer < \(b\) \((f_1 \neq 0)\)

**Note:** If \(x=0\), then \(e=0\) and all \(f_k=0\).

---

IBM Extension

XL Fortran implements this model with the following parameters:

<table>
<thead>
<tr>
<th>Real Kind parameter</th>
<th>(b) Parameter</th>
<th>(p) Parameter</th>
<th>(e_{\min}) Parameter</th>
<th>(e_{\max}) Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>24</td>
<td>-125</td>
<td>128</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>53</td>
<td>-1021</td>
<td>1024</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>106</td>
<td>-1021</td>
<td>1024</td>
</tr>
</tbody>
</table>

---

End of IBM Extension

The following intrinsic functions use this model:

- DIGITS
- EPSILON
- EXPONENT
- FRACTION
- HUGE
- MAXEXPONENT
- MINEXPONENT
- NEAREST
- PRECISION
- RADIX
- RANGE
- RRSPACING
- SCALE
- SET_EXPONENT
- SPACING
- TINY
- TRAP
- TRAPSUB
- TRAPTOP
- TRAPLEVEL
- TRAPBREAK
- TRAPDIST
- TRAPREG

---

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()**

---

**Purpose**
Terminates the program. It truncates all open output files to the current position of the file pointer, closes all open files, and sends the SIGABRT signal to the current process.

If this signal is neither caught nor ignored, the core file is saved in the file /cores/core.PID, where PID is the Process ID of the current process.

**Class**
Subroutine

**Examples**
The following is an example of a statement using the ABORT subroutine.

IF (ERROR_CONDITION) CALL ABORT

---

**ABS(A)**

**Purpose**
Absolute value.

**Class**
Elemental function
Argument type and attributes
A must be of type integer, real, or complex.

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.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>IABS</td>
<td>any integer</td>
<td>same as argument</td>
<td>yes</td>
</tr>
<tr>
<td>ABS</td>
<td>default real</td>
<td>default real</td>
<td>yes</td>
</tr>
<tr>
<td>DABS</td>
<td>double precision real</td>
<td>double precision real</td>
<td>yes</td>
</tr>
<tr>
<td>QABS</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
<td>yes</td>
</tr>
<tr>
<td>CABS</td>
<td>default complex</td>
<td>default real</td>
<td>yes</td>
</tr>
<tr>
<td>CDABS</td>
<td>double complex</td>
<td>double precision real</td>
<td>yes</td>
</tr>
<tr>
<td>ZABS</td>
<td>double complex</td>
<td>double precision real</td>
<td>yes</td>
</tr>
<tr>
<td>CQABS</td>
<td>COMPLEX(16)</td>
<td>REAL(16)</td>
<td>yes</td>
</tr>
</tbody>
</table>

Notes:
1. IBM Extension.
2. IBM Extension: the ability to specify a nondefault integer argument.

ACHAR(I)

Purpose
Returns the character in a specified position of the ASCII collating sequence. It is the inverse of the IACHAR function.

Class
Elemental function

Argument type and attributes
I must be of type integer.

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 $0 \leq I \leq 127$, 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)

Purpose

Arccosine (inverse cosine) function.

Class

Elemental function

Argument type and attributes

- $X$ must be of type real with a value that satisfies the inequality $|X| \leq 1$.

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.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACOS</td>
<td>default real</td>
<td>default real</td>
<td>yes</td>
</tr>
<tr>
<td>DACOS</td>
<td>double precision real</td>
<td>double precision real</td>
<td>yes</td>
</tr>
<tr>
<td>QACOS 1</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
<td>yes</td>
</tr>
<tr>
<td>QARCOS 1</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
<td>yes</td>
</tr>
</tbody>
</table>

Notes:

1. IBM Extension.

ACOSD(X)

Purpose

Arccosine (inverse cosine) function. Result in degrees.

Class

Elemental function
Argument type and attributes

\( X \) must be of type real. Its value must satisfy the inequality \(|X| \leq 1\).

Result type and attributes

Same as \( X \).

Result value

- It is expressed in degrees and approximates \( \arccos(X) \).
- It is in the range \( 0^\circ \leq ACOSD(X) \leq 180^\circ \).

Examples

\( ACOSD \ (0.5) \) has the value 60.0.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACOSD</td>
<td>default real</td>
<td>default real</td>
<td>yes</td>
</tr>
<tr>
<td>DACOSD</td>
<td>double precision real</td>
<td>double precision real</td>
<td>yes</td>
</tr>
<tr>
<td>QACOSD</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
<td>yes</td>
</tr>
</tbody>
</table>

ADJUSTL(STRING)

Purpose

Adjust to the left, removing leading blanks and inserting trailing blanks.

Class

Elemental function

Argument type and attributes

\( \text{STRING} \) must be of type character.

Result type and attributes

Character of the same length and kind type parameter as \( \text{STRING} \).

Result value

The value of the result is the same as \( \text{STRING} \) except that any leading blanks have been deleted and the same number of trailing blanks have been inserted.

Examples

\( \text{ADJUSTL ('bWORD')} \) has the value 'WORDb'.

ADJUSTR(STRING)

Purpose

Adjust to the right, removing trailing blanks and inserting leading blanks.
Class
Elemental function

Argument type and attributes
STRING must be of type character.

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 ('WORDb') has the value 'bWORD'.

AIMAG(Z), IMAG(Z)

Purpose
Imaginary part of a complex number.

Class
Elemental function

Argument type and attributes
Z must be of type complex.

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.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIMAG</td>
<td>default complex</td>
<td>default real</td>
<td>yes</td>
</tr>
<tr>
<td>DIMAG</td>
<td>double complex</td>
<td>double precision real</td>
<td>yes</td>
</tr>
<tr>
<td>QIMAG</td>
<td>COMPLEX(16)</td>
<td>REAL(16)</td>
<td>yes</td>
</tr>
</tbody>
</table>

Notes:
1. IBM Extension.
AINT(A, KIND)

Purpose
Truncates to a whole number.

Class
Elemental function

Argument type and attributes
A must be of type real.
KIND (optional)
must be a scalar integer initialization expression.

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| ≥ 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

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>AINT</td>
<td>default real</td>
<td>default real</td>
<td>yes</td>
</tr>
<tr>
<td>DINT</td>
<td>double precision real</td>
<td>double precision real</td>
<td>yes</td>
</tr>
<tr>
<td>QINT 1</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
<td>yes</td>
</tr>
</tbody>
</table>

Notes:
1. IBM Extension.

ALIGNX(K,M)

Purpose
The ALIGNX built-in subroutine enables you to assert the alignment of a variable at a certain point in the program flow. Specifically, at the call point to ALIGNX, you can assert that the remainder from dividing the address of the second argument by the value of the first argument is zero. In case the second argument is a Fortran 90 pointer, the assertion refers to the address of the target. In case the second argument is an integer pointer, the assertion refers to the address of the pointee. Should you give the compiler incorrect alignment, the resulting program may not run correctly.
Class
Subroutine

Argument type and attributes

K is an INTEGER(4) positive constant expression whose value is a power of two.

M is a variable of any type. When M is a Fortran 90 pointer, the pointer must be associated.

Examples

```
INTEGER*4 B(200)
DO N=1, 200
   CALL ALIGNX(4, B(N))  ! ASSERTS THAT AT THIS POINT, B(N) IS 4-BYTE ALIGNED
   B(N) = N
END DO
END

SUBROUTINE VEC(A, B, C)
   INTEGER A(200), B(200), C(200)
   CALL ALIGNX(16, A(1))
   CALL ALIGNX(16, B(1))
   CALL ALIGNX(16, C(1))
   DO N = 1, 200
      C(N) = A(N) + B(N)
   END DO
END SUBROUTINE
```

End of IBM Extension

---

ALL(MASK, DIM)

Purpose
Determines if all values in an entire array, or in each vector along a single dimension, are true.

Class
Transformational function

Argument type and attributes

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.

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 \((s_1, s_2, \ldots, s_{(\text{DIM}-1)}, s_{(\text{DIM}+1)}, \ldots, s_n)\), where \(n\) is the rank of MASK.

Each element in the result array is .TRUE. only if all the elements given by 
\[ \text{MASK}(m_1, m_2, \ldots, m_{(\text{DIM}-1)}, m_{(\text{DIM}+1)}, \ldots, m_n) \]
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 |
   | 2 4 1 |
! 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(X)**

**Purpose**

Indicates whether or not an allocatable object is currently allocated.

**Class**

Inquiry function

**Argument type and attributes**

\(X\) can be one of the following:

- **ARRAY** is an allocatable array whose allocation status you want to know.
- **SCALAR** is an allocatable scalar whose allocation status you want to know.

**Result type and attributes**

Default logical scalar.

**Result value**

The result corresponds to the allocation status of ARRAY or SCALAR: .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 `-qif90=autodealloc` compiler option there is no undefined allocation status.

**Examples**

```fortran
INTEGER, ALLOCATABLE, DIMENSION(:) :: A
PRINT *, ALLOCATED(A) ! A is not allocated yet.
ALLOCATE (A(1000))
PRINT *, ALLOCATED(A) ! A is now allocated.
END
```
ANINT(A, KIND)

**Purpose**
Nearest whole number.

**Class**
Elemental function

**Argument type and attributes**
- A must be of type real.
- KIND (optional)
  - must be a scalar integer initialization expression.

**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 ≤ 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

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANINT</td>
<td>default real</td>
<td>default real</td>
<td>yes</td>
</tr>
<tr>
<td>DNINT</td>
<td>double precision real</td>
<td>double precision real</td>
<td>yes</td>
</tr>
<tr>
<td>QNINT</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Notes:**
1. IBM Extension.

ANY(MASK, DIM)

**Purpose**
Determines if any of the values in an entire array, or in each vector along a single dimension, are true.

**Class**
Transformational function
Argument type and attributes

MASK is a logical array.

DIM (optional)

is an integer scalar in the range \(1 \leq \text{DIM} \leq \text{rank}(	ext{MASK})\). The corresponding actual argument must not be an optional dummy argument.

Result value

The result is a logical array of the same type and type parameters as \text{MASK}, and rank of \(\text{rank}(	ext{MASK})-1\). If the \text{DIM} is missing, or \text{MASK} has a rank of one, the result is a scalar of type logical.

The shape of the result is \((s_1, s_2, ..., s_{(\text{DIM}-1)}, s_{(\text{DIM}+1)}, ..., s_n)\), where \(n\) is the rank of \text{MASK}.

Each element in the result array is \(\text{.TRUE.}\) if any of the elements given by \text{MASK}(m_1, m_2, ..., m_{(\text{DIM}-1)}, m_{(\text{DIM}+1)}, ..., m_n) are true. When the result is a scalar, either because \text{DIM} is not specified or because \text{MASK} is of rank one, it is \(\text{.TRUE.}\) if any of the elements of \text{MASK} are true.

Examples

\[
\begin{array}{l}
! A \text{ is the array } 9 \ -6 \ 7 \ \text{, and } B \text{ is the array } 2 \ 7 \ 8 \ \\
! \begin{array}{l}
9 \ -6 \ 7 \\
3 \ -1 \ 5 \\
\end{array} \\
\begin{array}{l}
2 \ 7 \ 8 \\
5 \ 6 \ 9 \\
\end{array}
\end{array}
\]

! 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)

\textbf{ASIN(X)}

Purpose

Arcsine (inverse sine) function.

Class

Elemental function

Argument type and attributes

X must be of type real. Its value must satisfy the inequality \(|X| \leq 1\).

Result type and attributes

Same as X.

Result value

- It is expressed in radians, and approximates \(\text{arcsin}(X)\).
- It is in the range \(-\pi/2 \leq \text{ASIN}(X) \leq \pi/2\).
Examples

ASIN (1.0) approximates π/2.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASIN</td>
<td>default real</td>
<td>default real</td>
<td>yes</td>
</tr>
<tr>
<td>DASIN</td>
<td>double precision real</td>
<td>double precision real</td>
<td>yes</td>
</tr>
<tr>
<td>QASIN</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
<td>yes</td>
</tr>
<tr>
<td>QARSIN</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
<td>yes</td>
</tr>
</tbody>
</table>

Notes:
1. IBM Extension.

---

**ASIND(X)**

---

**IBM Extension**

**Purpose**

Arcsine (inverse sine) function. Result in degrees.

**Class**

Elemental function

**Argument type and attributes**

X must be of type real. Its value must satisfy the inequality |X| ≤ 1.

**Result type and attributes**

Same as X.

**Result value**

- It is expressed in degrees, and approximates arcsin(X).
- It is in the range -90° ≤ ASIND(X) ≤ 90°

**Examples**

ASIND (0.5) has the value 30.0.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASIND</td>
<td>default real</td>
<td>default real</td>
<td>yes</td>
</tr>
<tr>
<td>DASIND</td>
<td>double precision real</td>
<td>double precision real</td>
<td>yes</td>
</tr>
<tr>
<td>QASIND</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
<td>yes</td>
</tr>
</tbody>
</table>

---

**ASSOCIATED(POINTER, TARGET)**

---

**Purpose**

Returns the association status of its pointer argument, or indicates whether the pointer is associated with the target.
Class
Inquiry function

Argument type and attributes
POINTER A pointer or procedure 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, procedure pointer, or target that might or might not be associated with POINTER. Its association status must not be undefined.

Result type and attributes
Default logical attributes.

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).

If TARGET is present, then the result is .FALSE. if one of the following occurs:
  • POINTER is associated with a zero-sized array.
  • TARGET is associated with a zero-sized array.
  • 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)

Purpose
Arctangent (inverse tangent) function.
Class
Elemental function

Argument type and attributes
X must be of type real.

Result type and attributes
Same as X.

Result value
- It is expressed in radians and approximates arctan(X).
- It is in the range \(-\pi/2 \leq \text{ATAN}(X) \leq \pi/2\).

Examples
ATAN (1.0) approximates \(\pi/4\).

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATAN</td>
<td>default real</td>
<td>default real</td>
<td>yes</td>
</tr>
<tr>
<td>DATAN</td>
<td>double precision real</td>
<td>double precision real</td>
<td>yes</td>
</tr>
<tr>
<td>QATAN (\text{I})</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
<td>yes</td>
</tr>
</tbody>
</table>

Notes:
1. IBM Extension.

---

ATAND(X)

IBM Extension

Purpose
Arctangent (inverse tangent) function. Result in degrees.

Class
Elemental function

Argument type and attributes
X must be of type real.

Result type and attributes
Same as X.

Result value
- It is expressed in degrees and approximates arctan(X).
- It is in the range \(-90^\circ \leq \text{ATAND}(X) \leq 90^\circ\).

Examples
ATAND (1.0) has the value 45.0.
ATAN2(Y, X)

Purpose

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.

Class

Elemental function

Argument type and attributes

- **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.

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 ≠ 0, the result approximates \text{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 \(\pi\).
- If X = 0, the absolute value of the result is \(\pi/2\).

Examples

\text{ATAN2} (1.5574077, 1.0) has the value 1.0.

Given that:

\[
Y = \begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix} \quad X = \begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}
\]

the value of ATAN2(Y,X) is approximately:

\[
\text{ATAN2} \ (Y, X) = \begin{bmatrix} 3\pi/4 & \pi/4 \\ -3\pi/4 & -\pi/4 \end{bmatrix}
\]
Notes:
1. IBM Extension.

ATAN2D(Y, X)

IBM Extension

Purpose
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.

Class
Elemental function

Argument type and attributes
- 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.

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:

\[
Y = \begin{bmatrix}
1.0 & 1.0 \\
-1.0 & -1.0
\end{bmatrix} \quad X = \begin{bmatrix}
-1.0 & 1.0 \\
1.0 & -1.0
\end{bmatrix}
\]

then the value of ATAN2D(Y,X) is:

\[
ATAN2D(Y,X) = \begin{bmatrix}
135.0000000 & 45.0000000 \\
-135.0000000 & -45.0000000
\end{bmatrix}
\]

Specific Name | Argument Type | Result Type | Pass As Arg?
--- | --- | --- | ---
ATAN2D | default real | default real | yes
DATAN2D | double precision real | double precision real | yes
QATAN2D | REAL(16) | REAL(16) | yes
BIT_SIZE(I)

**Purpose**

Returns the number of bits in an integer type. Because only the type of the argument is examined, the argument need not be defined.

**Class**

Inquiry function

**Argument type and attributes**

| I | must be of type integer. |

**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:

<table>
<thead>
<tr>
<th>IBM Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>integer(1)</td>
</tr>
<tr>
<td>integer(2)</td>
</tr>
<tr>
<td>integer(4)</td>
</tr>
<tr>
<td>integer(8)</td>
</tr>
</tbody>
</table>

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)

**Purpose**

Tests a bit of an integer value.

**Class**

Elemental function

**Argument type and attributes**

| I | must be of type integer. |
| POS | must be of type integer. It must be nonnegative and be less than BIT_SIZE(I). |
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

\textbf{BTEST} (8, 3) has the value .TRUE.

If A has the value

\begin{verbatim}
  1 2
  3 4
\end{verbatim}

the value of BTEST (A, 2) is

\begin{verbatim}
false false
false true
\end{verbatim}

and the value of BTEST (2, A) is

\begin{verbatim}
true false
false false
\end{verbatim}

See “Integer bit model” on page 475.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTEST</td>
<td>any integer</td>
<td>default logical</td>
<td>yes</td>
</tr>
</tbody>
</table>

Notes:

1. IBM Extension.

\textbf{CEILING(A, KIND)}

Purpose

Returns the least integer greater than or equal to its argument.

Class

Elemental function

Argument type and attributes

A must be of type real.

KIND (optional)

must be a scalar integer initialization expression.

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 1001, with a kind type parameter of two.

CHAR(I, KIND)

Purpose
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.

Class
Elemental function

Argument type and attributes

I must be of type integer with a value in the range $0 \leq I \leq 127$.

KIND (optional)
must be a scalar integer initialization expression.

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

--- IBM Extension ---

CHAR (88) has the value ‘X’.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>any integer</td>
<td>default character</td>
<td>yes 1</td>
</tr>
</tbody>
</table>

Notes:
1. IBM Extension: the ability to specify a nondefault integer argument.
2. XL Fortran supports only the ASCII collating sequence.

--- End of IBM Extension ---

CMPLX(X, Y, KIND)

Purpose
Convert to complex type.

Class
Elemental function

Argument type and attributes
- 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.

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

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPLX</td>
<td>default real</td>
<td>default complex</td>
<td>no</td>
</tr>
</tbody>
</table>

Notes:

1. IBM Extension.

**Related information**

“DCMPLX(X, Y)” on page 507 “QCMPLX(X, Y)” on page 572

---

### COMMAND_ARGUMENT_COUNT()

**Purpose**

Returns the number of command line arguments for the command that invoked the program.

**Class**

Inquiry function

**Result type and attributes**

Default integer scalar

**Result value**

The result value is the number of command arguments, not counting the command name. If there are no command arguments, the result value is 0.

**Examples**

```fortran
integer cmd_count
cmd_count = COMMAND_ARGUMENT_COUNT()
prient*, cmd_count
end
```

The following is sample output generated by the above program:

$ a.out
0
$ a.out aa
1
$ a.out aa bb
2

End of Fortran 2003 Standard

---

### CONJG(Z)

**Purpose**

Conjugate of a complex number.

**Class**

Elemental function
Argument type and attributes

$Z$ must be of type complex.

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 $(2.0, -3.0)$.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONJG</td>
<td>default complex</td>
<td>default complex</td>
<td>yes</td>
</tr>
<tr>
<td>DCONJG</td>
<td>double complex</td>
<td>double complex</td>
<td>yes</td>
</tr>
<tr>
<td>QCONJG</td>
<td>COMPLEX(16)</td>
<td>COMPLEX(16)</td>
<td>yes</td>
</tr>
</tbody>
</table>

Notes:
1. IBM Extension.

COS(X)

Purpose

Cosine function.

Class

Elemental function

Argument type and attributes

$X$ must be of type real or complex.

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 and imaginary parts of $X$ are regarded as values in radians.

Examples

COS $(1.0)$ has the value $0.54030231$ (approximately).

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>COS</td>
<td>default real</td>
<td>default real</td>
<td>yes</td>
</tr>
<tr>
<td>Dcos</td>
<td>double precision real</td>
<td>double precision real</td>
<td>yes</td>
</tr>
<tr>
<td>QCOS</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
<td>yes</td>
</tr>
</tbody>
</table>
### Specific Name | Argument Type | Result Type | Pass As Arg?
---|---|---|---
CCOS | default complex | default complex | yes
CDCOS | double complex | double complex | yes
ZCOS | double complex | double complex | yes
CQCOS | COMPLEX(16) | COMPLEX(16) | yes

### Notes:
1. IBM Extension.
2. Given that X is a complex number in the form a + bi, where i = (-1)^{1/2}:
   a. abs(b) must be less than or equal to 88.7228; a is any real value.
   b. abs(b) must be less than or equal to 709.7827; a is any real value.

---

**COSD(X)**

**Purpose**
Cosine function. Argument in degrees.

**Class**
Elemental function

**Argument type and attributes**
X must be of type real.

**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)**

**Purpose**
Hyperbolic cosine function.
Class
Elemental function

Argument type and attributes
X must be of type real.

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).

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>COSH</td>
<td>default real</td>
<td>default real</td>
<td>yes</td>
</tr>
<tr>
<td>DCOSH</td>
<td>double precision real</td>
<td>double precision real</td>
<td>yes</td>
</tr>
<tr>
<td>QCOSH</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
<td>yes</td>
</tr>
</tbody>
</table>

Notes:
1. Given that X is a complex number in the form \( a + bi \), where \( i = (-1)^{\frac{1}{2}} \):
   a. \( \text{abs(b)} \) must be less than or equal to 88.7228; \( a \) is any real value.
   b. \( \text{abs(b)} \) must be less than or equal to 709.7827; \( a \) is any real value.
2. IBM Extension.

COUNT(MASK, DIM)

Purpose
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.

Class
Transformational function

Argument type and attributes

MASK is a logical array.

DIM (optional)
is an integer scalar in the range \( 1 \leq \text{DIM} \leq \text{rank(MASK)} \). The corresponding actual argument must not be an optional dummy argument.

Result value
If DIM is present, the result is an integer array of rank \( \text{rank(MASK)}-1 \). 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 \( (R(s_1, s_2, \ldots, s_{(DIM-1)}, s_{(DIM+1)}, \ldots, s_n)) \) equals the number of elements that are true in \textsc{mask} along the corresponding dimension \((s_1, s_2, \ldots, s_{(DIM-1)}, s_{(DIM+1)}, \ldots, s_n)\).

If \textsc{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 ]
! 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)**

*Fortran 95*

**Purpose**

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 \texttt{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.

**Class**

Subroutine

**Argument type and attributes**

<table>
<thead>
<tr>
<th>\textsc{time}</th>
<th>Is a scalar of type real. It is an \texttt{INTENT(OUT)} argument that is assigned an approximation to the processor time. The time is measured in seconds. The time returned by \texttt{CPU_TIME} is dependent upon the setting of the \textsc{xlf_rte_opts} environment variable run-time option \texttt{cpu_time_type}. The valid settings for \texttt{cpu_time_type} are:</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{usertime}</td>
<td>The user time for the current process.</td>
</tr>
<tr>
<td>\texttt{systime}</td>
<td>The system time for the current process.</td>
</tr>
<tr>
<td>\texttt{alltime}</td>
<td>The sum of the user and system time for the current process</td>
</tr>
<tr>
<td>\texttt{total_usertime}</td>
<td>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.</td>
</tr>
<tr>
<td>\texttt{total_systime}</td>
<td>The total system time for the current process. The</td>
</tr>
</tbody>
</table>
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.

**Examples**

**Example 1:**

```fortran
! 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:

```bash
export XLFRTEOPTS=cpu_time_type=usertime
```

**Example 2:**

```fortran
! 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 Compiler Reference for more information.

---

**CSHIFT(ARRAY, SHIFT, DIM)**

**Purpose**

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.
Class

Transformational function

Argument type and attributes

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.

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 (s₁, s₂, ..., s(DIM-1), s(DIM+1), ..., sₙ) is shifted according to the corresponding value in SHIFT (s₁, s₂, ..., s(DIM-1), s(DIM+1), ..., sₙ)

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)

IBM Extension

Purpose

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.

Class
Elemental function

Argument type and attributes

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSOURCE</td>
<td>is a scalar or array expression of type LOGICAL, INTEGER, or REAL and any kind except 1.</td>
</tr>
<tr>
<td>FSOURCE</td>
<td>is a scalar or array expression with the same type and type parameters as TSOURCE.</td>
</tr>
<tr>
<td>MASK</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.

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>
<thead>
<tr>
<th>Explanation</th>
<th>Function Return Value</th>
<th>Generic Name</th>
</tr>
</thead>
</table>
| 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        |

Table 31. Result values for CVMGx intrinsic procedures

End of IBM Extension
DATE_AND_TIME(DATE, TIME, ZONE, VALUES)

Purpose

Returns data from the real-time clock and the date in a form compatible with the representations defined in ISO 8601:1988.

Class

Subroutine

Argument type and attributes

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 expressed in minutes, respectively. If no clock is available, they are set to blank.

The value of ZONE may be incorrect if you have not set the time zone on your hardware correctly. You can manually set the TZ environment variable to ensure the time zone is correct.

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.

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.

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)

Purpose
Convert to double precision real type.

Class
Elemental function

Argument type and attributes
A must be of type integer, real, or complex.

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.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFLOAT</td>
<td>any integer</td>
<td>double precision real</td>
<td>no</td>
</tr>
<tr>
<td>DBLE</td>
<td>default real</td>
<td>double precision real</td>
<td>no</td>
</tr>
<tr>
<td>DBLEQ</td>
<td>REAL(16)</td>
<td>REAL(8)</td>
<td>no</td>
</tr>
</tbody>
</table>

End of IBM Extension

DCMPLX(X, Y)

Purpose

Convert to double complex type.

Class

Elemental function

Argument type and attributes

- 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.

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).
DIGITS(X)

**Purpose**

Returns the number of significant digits for numbers whose type and kind type parameter are the same as the argument.

**Class**

Inquiry function

**Argument type and attributes**

X must be of type integer or real. It may be scalar or array valued.

**Result type and attributes**

Default integer scalar.

**Result value**

---

**IBM Extension**

- If X is of type integer, the number of the significant digits of X is:

<table>
<thead>
<tr>
<th>type</th>
<th>bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer(1)</td>
<td>7</td>
</tr>
<tr>
<td>integer(2)</td>
<td>15</td>
</tr>
<tr>
<td>integer(4)</td>
<td>31</td>
</tr>
<tr>
<td>integer(8)</td>
<td>63</td>
</tr>
</tbody>
</table>

- If X is of type real, the number of significant bits of X is:

<table>
<thead>
<tr>
<th>type</th>
<th>bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>real(4)</td>
<td>24</td>
</tr>
<tr>
<td>real(8)</td>
<td>53</td>
</tr>
<tr>
<td>real(16)</td>
<td>106</td>
</tr>
</tbody>
</table>

---

**Examples**

---

**IBM Extension**

DIGITS(X) = 63, where X is of type integer(8) (see “Data representation models” on page 475).

---

End of IBM Extension
DIM(X, Y)

**Purpose**
The difference X-Y if it is positive; otherwise zero.

**Class**
Elemental function

**Argument type and attributes**
- **X** must be of type integer or real.
- **Y** must be of the same type and kind type parameter as **X**.

**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**

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDIM</td>
<td>any integer</td>
<td>same as argument</td>
<td>yes</td>
</tr>
<tr>
<td>DIM</td>
<td>default real</td>
<td>default real</td>
<td>yes</td>
</tr>
<tr>
<td>DDIM</td>
<td>double precision real</td>
<td>double precision real</td>
<td>yes</td>
</tr>
<tr>
<td>QDIM</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Notes:**
1. IBM Extension: the ability to specify a nondefault integer argument.
2. IBM Extension.

DOT_PRODUCT(VECTOR_A, VECTOR_B)

**Purpose**
Computes the dot product on two vectors.

**Class**
Transformational function

**Argument type and attributes**
- **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**.
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 11 on page 90 and Table 12 on page 94.

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)

Purpose

Double precision real product.

Class

Elemental function

Argument type and attributes

X must be of type default real.
Y must be of type default real.

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.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPROD</td>
<td>default real</td>
<td>double precision real</td>
<td>yes</td>
</tr>
<tr>
<td>QPROD</td>
<td>double precision real</td>
<td>REAL(16)</td>
<td>yes</td>
</tr>
</tbody>
</table>

Notes:
1. IBM Extension.
EOSHIFT(ARRAY, SHIFT, BOUNDARY, DIM)

**Purpose**
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.

**Class**
Transformational function

**Argument type and attributes**
- **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, **BOUNDARY** is a scalar or an expression of rank=rank(ARRAY)-1, and of shape (d1, d2..., dDIM-1, dDIM+1..., dn).
- **DIM** (optional) is an integer scalar in the range 1 ≤ DIM ≤ rank(ARRAY).

**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 (s₁, s₂, ..., s₁DIM, s₁DIM+1, ..., s₁n) are used for that dimension.
If BOUNDARY is not specified, the following default values are used, depending on the data type of ARRAY:

- **character**: 'b' (one blank)
- **logical**: false
- **integer**: 0
- **real**: 0.0
- **complex**: (0.0, 0.0)

**Examples**

```fortran
! A is | 1.1 4.4 7.7 |, SHIFT is
S=([-0.1, -0.2, -0.3]),
| 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)**

**Purpose**

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.

**Class**

Inquiry function

**Argument type and attributes**

- **X** must be of type real. It may be scalar or array valued.

**Result type and attributes**

Scalar of the same type and kind type parameter as X.

**Result value**

The result is

\[ 2.0 \times 10^{-\text{DIGITS}(X)} \]

where ei is the exponent indicator (E, D, or Q) depending on the type of X:
IBM Extension

<table>
<thead>
<tr>
<th>type</th>
<th>EPSILON(X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>real(4)</td>
<td>02E0 ** (-23)</td>
</tr>
<tr>
<td>real(8)</td>
<td>02D0 ** (-52)</td>
</tr>
<tr>
<td>real(16)</td>
<td>02Q0 ** (-105)</td>
</tr>
</tbody>
</table>

End of IBM Extension

Examples

IBM Extension

EPSILON(X) = 1.1920929E-07 for X of type real(4). See “Real data model” on page 477.

End of IBM Extension

ERF(X)

IBM Extension

Purpose

Error function.

\[
\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt
\]

Class

Elemental function

Argument type and attributes

\( X \) must be of type real.

Result type and attributes

Same as \( X \).

Result value

- The result value approximates \( \text{erf}(X) \).
- The result is in the range \(-1 \leq \text{ERF}(X) \leq 1\)

Examples

ERF (1.0) has the value 0.8427007794 (approximately).

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERF</td>
<td>default real</td>
<td>default real</td>
<td>yes</td>
</tr>
<tr>
<td>DERF</td>
<td>double precision real</td>
<td>double precision real</td>
<td>yes</td>
</tr>
<tr>
<td>QERF</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
<td>yes</td>
</tr>
</tbody>
</table>
**ERFC(X)**

**Purpose**
Complementary error function.

\[
\text{erfc}(x) = 1 - \text{erf}(x) = 2 \cdot \frac{1}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt
\]

**Class**
Elemental function

**Argument type and attributes**

\( X \) must be of type real.

**Result type and attributes**
Same as X.

**Result value**
- The result has a value equal to 1-\( \text{ERF}(X) \).
- The result is in the range \( 0 \leq \text{ERFC}(X) \leq 2 \)

**Examples**

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERFC</td>
<td>default real</td>
<td>default real</td>
<td>yes</td>
</tr>
<tr>
<td>DERFC</td>
<td>double precision real</td>
<td>double precision real</td>
<td>yes</td>
</tr>
<tr>
<td>QERFC</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
<td>yes</td>
</tr>
</tbody>
</table>

**EXP(X)**

**Purpose**
Exponential.

**Class**
Elemental function
**Argument type and attributes**

X must be of type real or complex.

**Result type and attributes**

Same as X.

**Result value**

- The result approximates $e^x$.
- If X is of type complex, its real and imaginary parts are regarded as values in radians.

**Examples**

EXP (1.0) has the value 2.7182818 (approximately).

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXP</td>
<td>default real</td>
<td>default real</td>
<td>yes</td>
</tr>
<tr>
<td>DEXP</td>
<td>double precision</td>
<td>double precision</td>
<td>yes</td>
</tr>
<tr>
<td>QEXP</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
<td>yes</td>
</tr>
<tr>
<td>CEXP</td>
<td>default complex</td>
<td>default complex</td>
<td>yes</td>
</tr>
<tr>
<td>CDEXP</td>
<td>double complex</td>
<td>double complex</td>
<td>yes</td>
</tr>
<tr>
<td>ZEXP</td>
<td>double complex</td>
<td>double complex</td>
<td>yes</td>
</tr>
<tr>
<td>CQEXP</td>
<td>COMPLEX(16)</td>
<td>COMPLEX(16)</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Notes:**

1. X must be less than or equal to 88.7228.
2. X must be less than or equal to 709.7827.
3. IBM Extension.
4. When X is a complex number in the form $a + bi$, where $i = (-1)^{1/2}$:
   a. $a$ must be less than or equal to 88.7228; $b$ is any real value.
   b. $a$ must be less than or equal to 709.7827; $b$ is any real value.

**EXPONENT(X)**

**Purpose**

Returns the exponent part of the argument when represented as a model number.

**Class**

Elemental function

**Argument type and attributes**

X must be of type real.

**Result type and attributes**

Default integer.
Result value

- If $X \neq 0$, 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

$\text{EXPONENT}(10.2) = 4$. See "Real data model" on page 477

FLOOR(A, KIND)

Purpose

Returns the greatest integer less than or equal to its argument.

Class

Elemental function

Argument type and attributes

- $A$ must be of type real.

KIND (optional)

- must be a scalar integer initialization expression.

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 greatest integer less than or equal to $A$.

- The result is undefined if the result cannot be represented as an integer of the specified KIND.

Examples

$\text{FLOOR}(-3.7)$ has the value -4.
$\text{FLOOR}(3.7)$ has the value 3.
FLOOR(1000.1, KIND=2) has the value 1000, with a kind type parameter of two.

FRACTION(X)

**Purpose**

Returns the fractional part of the model representation of the argument value.

**Class**

Elemental function

**Argument type and attributes**

X must be of type real.

**Result type and attributes**

Same as X.

**Result value**

The result is:

\[ X \times (2.0^{\text{EXPONENT}(X)}) \]

**Examples**

FRACTION(10.2) \times 2^{-4} \times 10.2 \text{ approximately equal to 0.6375}

GAMMA(X)

**Purpose**

Gamma function.

\[ \Gamma(x) = \int_0^{\infty} u^{x-1} e^{-u} du \]
Class
Elemental function

Argument type and attributes
X must be of type real.

Result type and attributes
Same as X.

Result value
The result has a value that approximates $\Gamma(X)$.

Examples
GAMMA (1.0) has the value 1.0.
GAMMA (10.0) has the value 362880.0 (approximately).

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAMMA</td>
<td>default real</td>
<td>default real</td>
<td>yes</td>
</tr>
<tr>
<td>DGamma</td>
<td>double precision real</td>
<td>double precision real</td>
<td>yes</td>
</tr>
<tr>
<td>QGamma</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
<td>yes</td>
</tr>
</tbody>
</table>

X must satisfy the inequality:
1. $-2.0^{*}23 < X \leq 35.0401$, except for nonpositive integral values
2. $-2.0^{*}52 < X \leq 171.6243$, except for nonpositive integral values
3. $-2.0^{*}105 < X \leq 171.6243$, except for nonpositive integral values

End of IBM Extension

GETENV(NAME, VALUE)

IBM Extension

Purpose
Returns the value of the specified environment variable.

Note: This is an IBM extension. It is recommended that you use the
GET_ENVIRONMENT_VARIABLE intrinsic procedure for portability.

Class
Subroutine

Argument type and attributes
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.

**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**

```fortran
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
```

GET_COMMAND(COMMAND, LENGTH, STATUS)

**Purpose**

Returns the command that invoked the program.

**Class**

Subroutine

**Argument type and attributes**

- **COMMAND (optional)**

  is the command that invoked the program, or a string of blanks if the command is unknown. COMMAND is an INTENT(OUT) argument that must be scalar of type default character.

- **LENGTH (optional)**

  is the significant length of the command that invoked the program, or 0 if the length of the command is unknown. This length includes significant trailing blanks of each argument. It does not include any truncation or padding that occurs when the command is assigned to the COMMAND argument. It is an INTENT(OUT) argument that must be scalar of type default integer.

- **STATUS (optional)**

  is a status value. It is an INTENT(OUT) argument that must be scalar of type default integer.
STATUS has one of the following values:

- 1 if the command retrieval fails
- -1 if the COMMAND argument is present and has a value less than the significant length of the command
- 0 otherwise

Examples

```fortran
integer len, status
character(7) :: cmd
call GET_COMMAND(cmd, len, status)
print*, cmd
print*, len
print*, status
end
```

The following is sample output the above program generates:

```
$ a.out
a.out   (followed by two spaces)
5
0
$ a.out aa
a.out a
8
-1
```

---

**GET_COMMAND_ARGUMENT(NUMBER, VALUE, LENGTH, STATUS)**

**Purpose**

Returns a command line argument of the command that invoked the program.

**Class**

Subroutine

**Argument type and attributes**

- **NUMBER** is an integer that identifies the argument number. 0 represents the command name. The numbers from 1 to the argument count represent the command’s arguments. It is an INTENT(IN) argument that must be scalar of type default integer.

- **VALUE** (optional) is assigned the value of the argument, or a string of blanks if the value is unknown. It is an INTENT(OUT) argument that must be scalar of type default character.

- **LENGTH** (optional) is assigned the significant length of the argument, or 0 if the length of the argument is unknown. This length includes significant trailing blanks. It does not include any truncation or padding that
occurs when the argument is assigned to the VALUE argument. It is an INTENT(OUT) argument that must be scalar of type default integer.

**STATUS (optional)**

is assigned a status value. It is an INTENT(OUT) argument that must be scalar of type default integer.

It has one of the following values:
- 1 if the argument retrieval fails
- -1 if the VALUE argument is present and has a value less than the significant length of the command argument
- 0 otherwise

**Examples**

```fortran
integer num, len, status
character*7 value
num = 0
call GET_COMMAND_ARGUMENT(num, value, len, status)
print*, value
print*, len
print*, status
```

The following is sample output generated by the above program:

```
$ a.out aa bb
a.out   (followed by two spaces)
5
0
```

End of Fortran 2003 Standard

---

**GET_ENVIRONMENT_VARIABLE(NAME, VALUE, LENGTH, STATUS, TRIM_NAME)**

**Purpose**

Returns the value of the specified environment variable.

**Class**

Subroutine

**Argument type and attributes**

**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 (optional)**

is the value of the environment variable, or a string of blanks if the environment variable has no value or does not exist. It is an INTENT(OUT) argument that must be scalar of type default character.
LENGTH (optional)
is the significant length of the value, or 0 if the environment variable has no value or does not exist. It is an INTENT(OUT) argument that must be scalar of type default integer.

STATUS (optional)
is a status value. It is an INTENT(OUT) argument that must be scalar of type default integer.

STATUS has one of the following values:
- 0, if either the environment variable exists and its value is succcesfully assigned to VALUE or the environment variable exists but has no value
- 1, if the environment variable does not exist
- -1, if the VALUE argument less than the significant length of value of the environment variable.
- 3, if other error conditions occur

TRIM_NAME (optional)
is a logical value that specifies whether to trim trailing blanks in NAME. By default, trailing blanks in NAME are trimmed. If TRIM_NAME exists and has the value .FALSE., trailing blanks in NAME are considered significant. TRIM_NAME is an INTENT(IN) argument that must be scalar of type logical.

Examples

```
integer num, len, status
character*15 value
call GET_ENVIRONMENT_VARIABLE('HOME', value, len, status)
print*, value
print*, len
print*, status
```

The following is sample output generated by the above program:

```
$ a.out
/home/xlfuser       (followed by two spaces)
13
0
```

HFIX(A)

IBM Extension

Purpose
Convert from REAL(4) to INTEGER(2).

This procedure is a specific function, not a generic function.

Class
Elemental function
Argument type and attributes

A must be of type REAL(4).

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.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>HFIX</td>
<td>REAL(4)</td>
<td>INTEGER(2)</td>
<td>no</td>
</tr>
</tbody>
</table>

--------- End of IBM Extension ---------

HUGE(X)

Purpose

Returns the largest number in the model representing numbers of the same type and kind type parameter as the argument.

Class

Inquiry function

Argument type and attributes

X must be of type integer or real. It may be scalar or array valued.

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:
  \[2^{\text{DIGITS}(X)} - 1\]
- If X is of any real type, the result is:
  \[
  (1.0 - 2.0^{-\text{DIGITS}(X)}) \times \\
  (2.0^{\text{MAXEXPONENT}(X)})
  \]

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 475.

IACHAR(C)

Purpose
Returns the position of a character in the ASCII collating sequence.

Class
Elemental function

Argument type and attributes
C must be of type default character and of length one.

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)

Purpose
Performs a bitwise AND on two integers.

Class
Elemental function

Argument type and attributes
I must be of type integer.
J must be of type integer with the same kind type parameter as I.

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:

<table>
<thead>
<tr>
<th>i_th bit of I</th>
<th>i_th bit of J</th>
<th>i_th bit of I(\land(I,J))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The bits are numbered from 0 to \(\text{BIT}_\text{SIZE}(I)-1\), from right to left.

**Examples**

I\(\land\) (1, 3) has the value 1. See “Integer bit model” on page 475.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>I(\land)</td>
<td>any integer</td>
<td>same as argument</td>
<td>yes</td>
</tr>
<tr>
<td>AND</td>
<td>any integer</td>
<td>same as argument</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Notes:**

1. IBM Extension.

---

**IBCLR(I, POS)**

**Purpose**

Clears one bit to zero.

**Class**

Elemental function

**Argument type and attributes**

I must be of type integer.

POS must be of type integer. It must be nonnegative and less than \(\text{BIT}_\text{SIZE}(I)\).

**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 \(\text{BIT}_\text{SIZE}(I)-1\), from right to left.

**Examples**

I\(\overline{\text{BCLR}}\) (14, 1) has the result 12.

If V has the value (/1, 2, 3, 4/), the value of I\(\overline{\text{BCLR}}\) (POS = V, I = 31) is (/29, 27, 23, 15/).
See “Integer bit model” on page 475.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBCLR I</td>
<td>any integer</td>
<td>same as argument</td>
<td>yes</td>
</tr>
</tbody>
</table>

Notes:
1. IBM Extension.

IBITS(I, POS, LEN)

**Purpose**
Extracts a sequence of bits.

**Class**
Elemental function

**Argument type and attributes**
- 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.

**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 475.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBITS I</td>
<td>any integer</td>
<td>same as argument</td>
<td>yes</td>
</tr>
</tbody>
</table>

Notes:
1. IBM Extension.

IBSET(I, POS)

**Purpose**
Sets one bit to one.

**Class**
Elemental function
Argument type and attributes

- I must be of type integer.
- POS must be of type integer. It must be nonnegative and less than BIT_SIZE (I).

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 475.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBSET</td>
<td>any integer</td>
<td>same as I</td>
<td>yes</td>
</tr>
</tbody>
</table>

Notes:

1. IBM Extension.

ICHAR(C)

Purpose

Returns the position of a character in the collating sequence associated with the kind type parameter of the character.

Class

Elemental function

Argument type and attributes

- C must be of type character and of length one. Its value must be that of a representable character.

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

--- IBM Extension ---

**ICHAR ("X")** has the value 88 in the ASCII collating sequence.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICHAR</td>
<td>default character</td>
<td>default integer</td>
<td>yes</td>
</tr>
</tbody>
</table>

Notes:
1. The extension is the ability to pass the name as an argument.
2. XL Fortran supports only the ASCII collating sequence.

--- End of IBM Extension ---

IEOR(I, J)

**Purpose**
Performs an exclusive OR.

**Class**
Elemental function

**Argument type and attributes**
- I must be of type integer.
- J must be of type integer with the same kind type parameter as I.

**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]th bit   [i]th bit   [i]th bit
of I       of J       of 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 475.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEOR I</td>
<td>any integer</td>
<td>same as argument</td>
<td>yes</td>
</tr>
<tr>
<td>XOR I</td>
<td>any integer</td>
<td>same as argument</td>
<td>yes</td>
</tr>
</tbody>
</table>
Notes:
1. IBM Extension.

ILEN(I)

Purpose
Returns one less than the length, in bits, of the two's complement representation of an integer.

Class
Elemental function

Argument type and attributes
I is of type integer

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)

Purpose
Identical to AIMAG.

Related information
“AIMAG(Z), IMAG(Z)” on page 482

INDEX(STRING, SUBSTRING, BACK)

Purpose
Returns the starting position of a substring within a string.
Class
Elemental function

Argument type and attributes

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.

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.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>default character</td>
<td>default integer</td>
<td>yes 1</td>
</tr>
</tbody>
</table>

Notes:
1. When this specific name is passed as an argument, the procedure can only be referenced without the back optional argument.

int(a, kind)

Purpose
Convert to integer type.

Class
Elemental function

Argument type and attributes

a must be of type integer, real, or complex.
kind (optional)
must be a scalar integer initialization expression.
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.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT</td>
<td>default real</td>
<td>default integer</td>
<td>no</td>
</tr>
<tr>
<td>IDINT</td>
<td>double precision real</td>
<td>default integer</td>
<td>no</td>
</tr>
<tr>
<td>IFIX</td>
<td>default real</td>
<td>default integer</td>
<td>no</td>
</tr>
<tr>
<td>IQINT</td>
<td>REAL(16)</td>
<td>default integer</td>
<td>no</td>
</tr>
</tbody>
</table>

Notes:
1. IBM Extension.

Related information

For information on alternative behavior for `INT` when porting programs to XL Fortran, see the `qport` compiler option in the `XL Fortran Compiler Reference`.

INT2(A)

IBM Extension

Purpose

Converts a real or integer value into a two byte integer.

Class

Elemental function

Argument type and attributes

A must be a scalar of integer or real type.

INT2 cannot be passed as an actual argument of another function call.

Result type and attributes

`INTEGER(2)` scalar
Result value

If \( A \) is of type integer, \( \text{INT2}(A) = A \).

If \( A \) is of type real, there are two possibilities:
- If \( |A| < 1 \), \( \text{INT2}(A) \) has the value 0
- If \( |A| \geq 1 \), \( \text{INT2}(A) \) 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 \( A \).

In both cases, truncation may occur.

Examples

The following is an example of the \( \text{INT2} \) function.

```fortran
REAL*4 :: R4
REAL*8 :: R8
INTEGER*4 :: I4
INTEGER*8 :: I8

R4 = 8.8; R8 = 18.9
I4 = 4; I8 = 8
PRINT *, INT2(R4), INT2(R8), INT2(I4), INT2(I8)
PRINT *, INT2(2.3), INT2(6)
PRINT *, INT2(65535.78), INT2(65536.89)
END
```

The following is sample output generated by the program above:

```
8 18 4 8
2 6
-1 0
```

! The results indicate that truncation has occurred, since ! only the last two bytes were saved.

End of IBM Extension

IOR(I, J)

Purpose

Performs an inclusive OR.

Class

Elemental function

Argument type and attributes

- \( I \) must be of type integer.
- \( J \) must be of type integer with the same kind type parameter as \( I \).

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:
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 475.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOR</td>
<td>any integer</td>
<td>same as argument</td>
<td>yes</td>
</tr>
<tr>
<td>OR</td>
<td>any integer</td>
<td>same as argument</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Notes:**
1. IBM Extension.

**ISHFT(I, SHIFT)**

**Purpose**

Performs a logical shift.

**Class**

Elemental function

**Argument type and attributes**

- **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).

**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 475.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISHFT</td>
<td>any integer</td>
<td>same as argument</td>
<td>yes</td>
</tr>
</tbody>
</table>
ISHFTC(I, SHIFT, SIZE)

**Purpose**
Performs a circular shift of the rightmost bits; that is, bits shifted off one end are inserted again at the other end.

**Class**
Elemental function

**Argument type and attributes**
- **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).

**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 475.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISHFTC</td>
<td>any integer</td>
<td>same as argument</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Notes:**
1. When this specific name is passed as an argument, the procedure can only be referenced with all three arguments.
KIND(X)

**Purpose**
Returns the value of the kind type parameter of X.

**Class**
Inquiry function

**Argument type and attributes**
X may be of any intrinsic type.

**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 16.

**Examples**
KIND (0.0) has the kind type parameter value of the default real type.

LBOUND(ARRAY, DIM)

**Purpose**
Returns the lower bound of each dimension in an array, or the lower bound of a specified dimension.

**Class**
Inquiry function

**Argument type and attributes**

<table>
<thead>
<tr>
<th>ARRAY</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIM (optional)</td>
<td>is an integer scalar in the range 1 ≤ DIM ≤ rank(ARRAY). The corresponding actual argument must not be an optional dummy argument.</td>
</tr>
</tbody>
</table>

**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**

```fortran
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)**

---

**IBM Extension**

**Purpose**

Returns the number of leading zero-bits in the binary representation of an integer.

**Class**

Elemental function

**Argument type and attributes**

- `I` must be of type integer.

**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**

```fortran
I = LEADZ(0_4) ! I=32
J = LEADZ(4_4) ! J=29
K = LEADZ(-I_4) ! K=0
```

---

End of IBM Extension
LEN(STRING)

Purpose
Returns the length of a character entity. The argument to this function need not be defined.

Class
Inquiry function

Argument type and attributes
STRING must be of type character. It may be scalar or array valued.

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.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEN</td>
<td>default character</td>
<td>default integer</td>
<td>yes</td>
</tr>
</tbody>
</table>

Notes:
1. IBM Extension: the ability to pass the name as an argument.

LEN_TRIM(STRING)

Purpose
Returns the length of the character argument without counting trailing blank characters.

Class
Elemental function

Argument type and attributes
STRING must be of type character.

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 ('bAbBb') has the value 4. LEN_TRIM ('bb') has the value 0.

---

**LGAMMA(X)**

| IBM Extension |

**Purpose**

Log of gamma function.

\[ \log_e \Gamma(x) = \log_e \int_0^\infty u^{x-1}e^{-u}du \]

**Class**

Elemental function

**Argument type and attributes**

X must be of type real.

**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).

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>LGAMMA</td>
<td>default real</td>
<td>default real</td>
<td>no</td>
</tr>
<tr>
<td>LGAMMA</td>
<td>double precision</td>
<td>double precision real</td>
<td>no</td>
</tr>
<tr>
<td>ALGAMA</td>
<td>default real</td>
<td>default real</td>
<td>yes</td>
</tr>
<tr>
<td>DLGAMA</td>
<td>double precision</td>
<td>double precision real</td>
<td>yes</td>
</tr>
<tr>
<td>QLGAMA</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
<td>yes</td>
</tr>
</tbody>
</table>

**X must satisfy the inequality:**

1. \( 0 < X \leq 4.0850E36 \).
2. \( 2.3561D-304 \leq X \leq 2^{1014} \).
3. \( 2.3561Q-304 \leq X \leq 2^{1014} \).
LGE(STRING_A, STRING_B)

Purpose
Test whether a string is lexically greater than or equal to another string, based on the ASCII collating sequence.

Class
Elemental function

Argument type and attributes
- STRING_A must be of type default character.
- STRING_B must be of type default character.

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..

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>LGE</td>
<td>default character</td>
<td>default logical</td>
<td>yes</td>
</tr>
</tbody>
</table>

Notes:
1. IBM Extension: the ability to pass the name as an argument.

LGT(STRING_A, STRING_B)

Purpose
Test whether a string is lexically greater than another string, based on the ASCII collating sequence.

Class
Elemental function

Argument type and attributes
- STRING_A must be of type default character.
STRING_B must be of type default character.

**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..

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>LGT</td>
<td>default character</td>
<td>default logical</td>
<td>yes 1</td>
</tr>
</tbody>
</table>

**Notes:**
1. IBM Extension: the ability to pass the name as an argument.

---

**LLE(STRING_A, STRING_B)**

**Purpose**
Test whether a string is lexically less than or equal to another string, based on the ASCII collating sequence.

**Class**
Elemental function

**Argument type and attributes**
- STRING_A must be of type default character.
- STRING_B must be of type default character.

**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..  

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLE</td>
<td>default character</td>
<td>default logical</td>
<td>yes ✓</td>
</tr>
</tbody>
</table>

Notes:
1. IBM Extension: the ability to pass the name as an argument.

LLT(STRING_A, STRING_B)

Purpose
Test whether a string is lexically less than another string, based on the ASCII collating sequence.

Class
Elemental function

Argument type and attributes

<table>
<thead>
<tr>
<th>Argument Type</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>STRING_A</td>
<td>must be of type default character.</td>
</tr>
<tr>
<td>STRING_B</td>
<td>must be of type default character.</td>
</tr>
</tbody>
</table>

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..

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLT</td>
<td>default character</td>
<td>default logical</td>
<td>yes ✓</td>
</tr>
</tbody>
</table>

Notes:
1. IBM Extension: the ability to pass the name as an argument.
**LOC(X)**

**IBM Extension**

**Purpose**

Returns the address of X that can then be used to define an integer POINTER.

**Class**

Inquiry function

**Argument type and attributes**

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.

**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**

```fortran
INTEGER A,B
POINTER (P,I)
P=LOC(A)
P=LOC(B)
END
```

------------------- End of IBM Extension -------------------

**LOG(X)**

**Purpose**

Natural logarithm.

**Class**

Elemental function

**Argument type and attributes**

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.
Result type and attributes
Same as X.

Result value
- It has a value approximating \( \log_e X \).
- For complex arguments, \( \text{LOG}((a, b)) \) approximates \( \text{LOG}(\text{ABS}((a, b))) + \text{ATAN2}((b, a)) \).

If the argument type is complex, the result is the principal value of the imaginary part \( \omega \) in the range \(-\pi < \omega \leq \pi \). If the real part of the argument is less than zero and its imaginary part is zero, the imaginary part of the result approximates \( \pi \).

Examples
\( \text{LOG} (10.0) \) has the value 2.3025851 (approximately).

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALOG</td>
<td>default real</td>
<td>default real</td>
<td>yes</td>
</tr>
<tr>
<td>DLOG</td>
<td>double precision real</td>
<td>double precision real</td>
<td>yes</td>
</tr>
<tr>
<td>QLOG</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
<td>yes</td>
</tr>
<tr>
<td>CLOG</td>
<td>default complex</td>
<td>default complex</td>
<td>yes</td>
</tr>
<tr>
<td>CDLOG</td>
<td>double complex</td>
<td>double complex</td>
<td>yes</td>
</tr>
<tr>
<td>ZLOG</td>
<td>double complex</td>
<td>double complex</td>
<td>yes</td>
</tr>
<tr>
<td>CQLOG</td>
<td>COMPLEX(16)</td>
<td>COMPLEX(16)</td>
<td>yes</td>
</tr>
</tbody>
</table>

Notes:
1. IBM Extension: the ability to pass the name as an argument.

LOG10(X)

Purpose
Common logarithm.

Class
Elemental function

Argument type and attributes
\( X \) must be of type real. The value of \( X \) must be greater than zero.

Result type and attributes
Same as \( X \).

Result value
The result has a value equal to \( \log_{10} X \).

Examples
\( \text{LOG10} (10.0) \) has the value 1.0.
<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALOG10</td>
<td>default real</td>
<td>default real</td>
<td>yes</td>
</tr>
<tr>
<td>DLOG10</td>
<td>double precision</td>
<td>double precision</td>
<td>yes</td>
</tr>
<tr>
<td>QLOG10</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
<td>yes</td>
</tr>
</tbody>
</table>

Notes:
1. IBM Extension: the ability to pass the name as an argument.

LOGICAL(L, KIND)

**Purpose**
Converts between objects of type logical with different kind type parameter values.

**Class**
Elemental function

**Argument type and attributes**
- **L** must be of type logical.
- **KIND (optional)** must be a scalar integer initialization expression.

**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)

**Purpose**
Performs a logical shift to the left.

**Class**
Elemental function

**Argument type and attributes**
- **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).
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.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSHIFT</td>
<td>any integer</td>
<td>same as argument</td>
<td>yes</td>
</tr>
</tbody>
</table>

End of IBM Extension

MATMUL(MATRIX_A, MATRIX_B, MINDIM)

Purpose
Performs a matrix multiplication.

Class
Transformational function

Argument type and attributes

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.

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:
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 libpthreads.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, real, or complex
and have the same kind.
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.

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 11 on page 90 and Table 12 on page 94.

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 |
| 4 5 6 |

! B is the array |
| 7 10 |
| 8 11 |
| 9 12 |

RES = MATMUL(A, B)
The result is | 50 68 |
| 122 167 |

IBM Extension

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)

End of IBM Extension

Related information

The numerical stability of Strassen’s method for matrix multiplication is discussed in:

End of IBM Extension

MAX(A1, A2, A3, ...)

Purpose
Maximum value.

Class
Elemental function

Argument type and attributes

• 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.

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.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMAX0</td>
<td>any integer</td>
<td>default real</td>
<td>no</td>
</tr>
<tr>
<td>AMAX1</td>
<td>default real</td>
<td>default real</td>
<td>no</td>
</tr>
<tr>
<td>DMAX1</td>
<td>double precision real</td>
<td>double precision real</td>
<td>no</td>
</tr>
<tr>
<td>QMAX1</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
<td>no</td>
</tr>
<tr>
<td>MAX0</td>
<td>any integer</td>
<td>same as argument</td>
<td>no</td>
</tr>
<tr>
<td>MAX1</td>
<td>any real</td>
<td>default integer</td>
<td>no</td>
</tr>
</tbody>
</table>

Notes:
1. IBM Extension: the ability to specify a nondefault integer argument.
2. IBM Extension: the ability to specify a nondefault real argument.

MAXEXPONENT(X)

Purpose

Returns the maximum exponent in the model representing numbers of the same type and kind type parameter as the argument.

Class

Inquiry function

Argument type and attributes

X must be of type real. It may be scalar or array valued.

Result type and attributes

Default integer scalar.

Result value

The result is the following:

<table>
<thead>
<tr>
<th>type</th>
<th>MAXEXPONENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>real(4)</td>
<td>128</td>
</tr>
<tr>
<td>real(8)</td>
<td>1024</td>
</tr>
<tr>
<td>real(16)</td>
<td>1024</td>
</tr>
</tbody>
</table>

End of IBM Extension
Examples

IBM Extension

MAXEXPONENT(X) = 128 for X of type real(4).

See “Real data model” on page 477.

End of IBM Extension

MAXLOC(ARRAY, DIM, MASK) or MAXLOC(ARRAY, MASK)

Purpose
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.

Class
Transformational function

Argument type and attributes

ARRAY is an array of type integer or real.

DIM (optional) 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.

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 (s1, ..., sDIM-1, sDIM+1, ..., sn), 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.

---

**Fortran 95**

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 |  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)**

**Purpose**

Returns the maximum value of the elements in the array along a dimension corresponding to the true elements of **MASK**.

**Class**

Transformational function

**Argument type and attributes**

- **ARRAY** is an array of type integer or real.
DIM (optional)
is an integer scalar in the range $1 \leq \text{DIM} \leq \text{rank(ARRAY)}$.

MASK (optional)
is an array or scalar of type logical that conforms to \texttt{ARRAY} in shape. If it is absent, the entire array is evaluated.

**Result value**
The result is an array of rank \text{rank(ARRAY)}-1, with the same data type as \texttt{ARRAY}. If \text{DIM} is missing or if \texttt{ARRAY} is of rank one, the result is a scalar.

If \text{DIM} is specified, each element of the result value contains the maximum value of all the elements that satisfy the condition specified by \texttt{MASK} along each vector of the dimension \text{DIM}. The array element subscripts in the result are $(s_1, s_2, \ldots, s_{(\text{DIM}-1)}, s_{(\text{DIM}+1)}, \ldots, s_n)$, where \text{n} is the rank of \texttt{ARRAY} and \text{DIM} is the dimension specified by \text{DIM}.

If \text{DIM} is not specified, the function returns the maximum value of all applicable elements.

If \texttt{ARRAY} is zero-sized or the mask array has all \text{.FALSE.} values, the result value is the negative number of the largest magnitude, of the same type and kind type as \texttt{ARRAY}.

---

**Fortran 95**

Because both \texttt{DIM} and \texttt{MASK} are optional, various combinations of arguments are possible. When the \texttt{-qintlog} option is specified with two arguments, the second argument refers to one of the following:

- \texttt{MASK} if it is an array of type integer, logical, byte or typeless
- \texttt{DIM} if it is a scalar of type integer, byte or typeless
- \texttt{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)

Purpose
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.

Class
Elemental function

Argument type and attributes
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.

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
!
! 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, ...)

Purpose
Minimum value.
Class

Elemental function

Argument type and attributes

- \( A_3, \ldots \) 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.

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

\( \text{MIN} (-9.0, 7.0, 2.0) \) has the value -9.0.

If you evaluate \( \text{MIN} (10, 3, A) \), where \( A \) is an optional array argument in the calling procedure, \( \text{PRESENT}(A) \) must be true in the calling procedure.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMIN0</td>
<td>any integer</td>
<td>default real</td>
<td>no</td>
</tr>
<tr>
<td>AMIN1</td>
<td>default real</td>
<td>default real</td>
<td>no</td>
</tr>
<tr>
<td>DMIN1</td>
<td>double precision real</td>
<td>double precision real</td>
<td>no</td>
</tr>
<tr>
<td>QMIN1</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
<td>no</td>
</tr>
<tr>
<td>MIN0</td>
<td>any integer</td>
<td>same as argument</td>
<td>no</td>
</tr>
<tr>
<td>MIN1</td>
<td>any real</td>
<td>default integer</td>
<td>no</td>
</tr>
</tbody>
</table>

\( \text{MINEXPONENT}(X) \)

Purpose

Returns the minimum (most negative) exponent in the model representing the numbers of the same type and kind type parameter as the argument.

Class

Inquiry function

Argument type and attributes

\( X \) must be of type real. It may be scalar or array valued.

Result type and attributes

Default integer scalar.
**Result value**

IBM Extension

The result is the following:

<table>
<thead>
<tr>
<th>type</th>
<th>MINEXPONENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>real(4)</td>
<td>-125</td>
</tr>
<tr>
<td>real(8)</td>
<td>-1021</td>
</tr>
<tr>
<td>real(16)</td>
<td>-968</td>
</tr>
</tbody>
</table>

---

**Examples**

IBM Extension

MINEXPONENT(X) = -125 for X of type real(4).

See "Real data model" on page 477.

---

**MINLOC(ARRAY, DIM, MASK) or MINLOC(ARRAY, MASK)**

**Purpose**

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.

**Class**

Transformational function

**Argument type and attributes**

**ARRAY**

is an array of type integer or real.

---

**DIM (optional)**

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.

**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 (s1, ..., sDIM−1, sDIM+1, ..., sn), 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.

---

**Fortran 95**

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

---

**Examples**

```fortran
! 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:

```fortran
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)

**Purpose**

Returns the minimum value of the elements in the array along a dimension corresponding to the true elements of MASK.

**Class**

Transformational function

**Argument type and attributes**

- **ARRAY** is an array of type integer or real.
- **DIM** (optional) is an integer scalar in the range $1 \leq \text{DIM} \leq \text{rank(ARRAY)}$.
- **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.

**Result value**

The result is an array of rank $\text{rank(ARRAY)}-1$, 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 $(s_1, s_2, ..., s_{(\text{DIM}-1)}, s_{\text{DIM+1}}, ..., s_n)$, 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.

---

**End of Fortran 95**

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)**

**Purpose**
Remainder function.

**Class**
Elemental function

**Argument type and attributes**
- **A** must be of type integer or real.
- **P** must be of the same type and kind type parameter as A.

---

**IBM Extension**
The kind type parameters can be different if the compiler option -qport=mod is specified.

---

**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.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOD</td>
<td>any integer</td>
<td>same as argument</td>
<td>yes</td>
</tr>
<tr>
<td>AMOD</td>
<td>default real</td>
<td>default real</td>
<td>yes</td>
</tr>
<tr>
<td>Specific Name</td>
<td>Argument Type</td>
<td>Result Type</td>
<td>Pass As Arg?</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------</td>
<td>---------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>DMOD</td>
<td>double precision real</td>
<td>double precision real</td>
<td>yes</td>
</tr>
<tr>
<td>QMOD</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Notes:**
1. IBM Extension: the ability to pass the name as an argument.

**Related information**
For information on alternative behavior for **MOD** when porting programs to XL Fortran, see the [qport] compiler option in the [XL Fortran Compiler Reference].

---

### MODULO(A, P)

**Purpose**
Modulo function.

**Class**
Elemental function

**Argument type and attributes**
- A must be of type integer or real.
- P must be of the same type and kind type parameter as A.

**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 ≤ R < P hold.
  - If P < 0, P < R ≤ 0 hold.
  - If P = 0, the result is undefined.
- Case (ii): A is of type real. If P ≠ 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)

**Purpose**
Copies a sequence of bits from one data object to another.
Class
Elemental subroutine

Argument type and attributes
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).

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 475.

NEAREST(X,S)

Purpose
Returns the nearest different processor-representable number in the direction indicated by the sign of S (toward positive or negative infinity).

Class
Elemental function

Argument type and attributes
X must be of type real.
S must be of type real and not equal to zero.

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**

<table>
<thead>
<tr>
<th>IBM Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEAREST $(3.0, 2.0) = 3.0 + 2.0^{(-22)}$. See “Real data model” on page 477.</td>
</tr>
</tbody>
</table>

**NEW_LINE(A)**

**Fortran 2003 Standard**

**Purpose**

The `NEW_LINE` intrinsic returns a new line character.

**Class**

Inquiry function

**Argument type and attributes**

$A$ must be a scalar or an array of type `character`

**Result type and attributes**

Character scalar of length one.

**Result value**

The result is the same as `ACHAR(10)`.

**Examples**

The following example uses the `NEW_LINE` intrinsic in list-directed output:

```fortran
character(1) c
print *, 'The first sentence.', NEW_LINE(c), 'The second sentence.'
```

Expected Output:

The first sentence.
The second sentence.

The following example passes a character literal constant to the `NEW_LINE` intrinsic:

```fortran
character(100) line
line = 'IBM' // NEW_LINE('Fortran') // 'XL Fortran Compiler'
```

Expected Output:

IBM
XL Fortran Compiler
NINT(A, KIND)

Purpose
Nearest integer.

Class
Elemental function

Argument type and attributes
A must be of type real.
KIND (optional) must be a scalar integer initialization expression.

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 \( A \leq 0 \), 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.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>NINT</td>
<td>default real</td>
<td>default integer</td>
<td>yes</td>
</tr>
<tr>
<td>IDNINT</td>
<td>double precision real</td>
<td>default integer</td>
<td>yes</td>
</tr>
<tr>
<td>IQNINT</td>
<td>REAL(16)</td>
<td>default integer</td>
<td>yes [1]</td>
</tr>
</tbody>
</table>

Notes:
1. IBM Extension: the ability to pass the name as an argument.

NOT(I)

Purpose
Performs a bitwise complement of integer.

Class
Elemental function

Argument type and attributes
I must be of type integer.
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:

<table>
<thead>
<tr>
<th>( i )th bit of ( I )</th>
<th>( i )th bit of ( \text{NOT} ,(I) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 1 )</td>
<td>( 0 )</td>
</tr>
<tr>
<td>( 0 )</td>
<td>( 1 )</td>
</tr>
</tbody>
</table>

The bits are numbered 0 to \( \text{BIT\_SIZE}(I)-1 \), from right to left.

Examples
If \( I \) is represented by the string of bits 01010101, \( \text{NOT} \,(I) \) has the string of bits 10101010. See “Integer bit model” on page 475.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{NOT} )</td>
<td>any integer</td>
<td>same as argument</td>
<td>yes ( \text{I} )</td>
</tr>
</tbody>
</table>

Notes:
1. IBM Extension.

NULL(MOLD)

Purpose
This function returns a pointer or designates an unallocated allocatable component of a structure constructor. The association status of the pointer is disassociated.

You must use the function without the \( \text{MOLD} \) argument in any of the following:
- initialisation of an object in a declaration
- default initialization of a component
- in a \( \text{DATA} \) statement
- in a \( \text{STATIC} \) statement

You can use the function with or without the \( \text{MOLD} \) argument in any of the following:
- in the \( \text{PARAMETER} \) attribute
- on the right side of a pointer assignment
- in a structure constructor
- as an actual argument

Class
Transformational function.
Argument type and attributes

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.

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 entity’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 or an unallocated allocatable entity.

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())

End of Fortran 95

NUM_PARTHDS()

IBM Extension

Purpose

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 `-qsm` has not been specified, then `NUM_PARTHDS` will always return a value of 1.

**Examples**

```fortran
I = NUM_PARTHDS()
IF (I == 1) THEN
    CALL SINGLE_THREAD_ROUTINE()
ELSE
    CALL MULTI_THREAD_ROUTINE()
```

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUM_PARTHDS</td>
<td>default scalar integer</td>
<td>default scalar integer</td>
<td>no</td>
</tr>
</tbody>
</table>

**Related information**
See the `parthds` and `XLSMPOPTS` runtime options in the [XL Fortran Optimization and Programming Guide](#).

---

**NUMBER_OF_PROCESSORS(DIM)**

---

**IBM Extension**

**Purpose**
Returns a scalar of type default integer whose value is always 1. This intrinsic ensures compatibility with programs written for High Performance Fortran (HPF) environments.

**Class**
System inquiry function

**Argument type and attributes**

DIM (optional)

must be a scalar integer and have a value of 1 (the rank of the processor array).

**Result type and attributes**
Default scalar integer which always has a value of 1.

**Examples**

```fortran
I = NUMBER_OF_PROCESSORS() ! 1
J = NUMBER_OF_PROCESSORS(DIM=1) ! 1
```
NUM_USRTHDS()

Purpose
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.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUM_USRTHDS</td>
<td>default scalar integer</td>
<td>default scalar integer</td>
<td>no</td>
</tr>
</tbody>
</table>

Related information
See the USRTHDS and the XLSMOPTS runtime options in the XL Fortran Optimization and Programming Guide.

PACK(ARRAY, MASK, VECTOR)

Purpose
Takes some or all elements from an array and packs them into a one-dimensional array, under the control of a mask.

Class
Transformational function

Argument type and attributes

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 and type parameter 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.

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.

POPCNT(I)

IBM Extension

Purpose
Population count.

Counts the number of set bits in a data object.

Class
Elemental function.

Argument type and attributes

I An INTENT(IN) argument of type BYTE, INTEGER, LOGICAL, or REAL.
If the argument is of type REAL, it must not be REAL(16).

Result type and attributes

Default integer.
Result value

The number of bits set to ON or 1.

Examples

<table>
<thead>
<tr>
<th>INTEGER</th>
<th>BIT REPRESENTATION</th>
<th>POPCNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>1</td>
</tr>
</tbody>
</table>

Related information

Data representation models

End of IBM Extension

POPPAR(I)

IBM Extension

Purpose

Population parity.

Determines the parity for a data object.

Class

Elemental function.

Argument type and attributes

I An INTENT(IN) argument of type BYTE, INTEGER, LOGICAL, or REAL. If the argument is of type REAL, it must not be REAL(16).

Result type and attributes

Default integer.

Result value

Returns 1 if there are an odd number of bits set.

Returns 0 if there are an even number of bits set.

Examples

<table>
<thead>
<tr>
<th>INTEGER</th>
<th>BIT REPRESENTATION</th>
<th>POPPAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>1</td>
</tr>
</tbody>
</table>
Related information
Data representation models

PRECISION(X)

Purpose
Returns the decimal precision in the model representing real numbers with the same kind type parameter as the argument.

Class
Inquiry function

Argument type and attributes
X must be of type real or complex. It may be scalar or array valued.

Result type and attributes
Default integer scalar.

Result value
The result is:

\[
\text{INT}( (\text{DIGITS}(X) - 1) \times \log_{10}(2) )
\]

Therefore,

<table>
<thead>
<tr>
<th>Type</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{real}(4), \text{complex}(4)</td>
<td>6</td>
</tr>
<tr>
<td>\text{real}(8), \text{complex}(8)</td>
<td>15</td>
</tr>
<tr>
<td>\text{real}(16), \text{complex}(16)</td>
<td>31</td>
</tr>
</tbody>
</table>

Examples

\[
\text{PRECISION}(X) = \text{INT}( (24 - 1) \times \log_{10}(2) ) = \text{INT}(6.92 \ldots) = 6 \text{ for } X \text{ of type real}(4). \text{ See } \text{"Real data model" on page 477.}
\]

End of IBM Extension
PRESENT(A)

Purpose
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.

Class
Inquiry function

Argument type and attributes
A is the name of an optional dummy argument that is accessible in the procedure in which the PRESENT function reference appears.

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
ENDIF
END SUBROUTINE SUB

Related information
“OPTIONAL” on page 360

PROCESSORS_SHAPE()

Purpose
Returns a zero-sized array. This intrinsic ensures compatibility with programs written for High Performance Fortran (HPF) environments.

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**
```fortran
I=PROCESSORS Shape()
! Zero-sized vector of type default integer
```

---

**PRODUCT(ARRAY, DIM, MASK) or PRODUCT(ARRAY, MASK)**

**Purpose**
Multiplies together all elements in an entire array, or selected elements from all vectors in a specified dimension of an array.

**Class**
Transformational function

**Argument type and attributes**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRAY</td>
<td>is an array with a numeric data type.</td>
</tr>
<tr>
<td>DIM (optional)</td>
<td>is an integer scalar (a specified dimension of ARRAY) in the range 1 ≤ DIM ≤ rank(ARRAY).</td>
</tr>
<tr>
<td>MASK (optional)</td>
<td>is a logical expression that conforms with ARRAY in shape. If MASK is a scalar, the scalar value applies to all elements in ARRAY.</td>
</tr>
</tbody>
</table>

**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 and ARRAY has a rank of one, the result is a scalar equal to the product of all elements of ARRAY that have a corresponding .TRUE. array element in MASK.

If DIM is also specified and ARRAY has rank greater than one, the result is a new array in which dimension DIM has been eliminated. Each new array element is the product of elements from a corresponding vector within ARRAY. The index values of that vector, in all dimensions except DIM, match those of the output element. The output element is the product of those vector elements 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.
  
  ! Multiply all elements in an array.
  ! RES = PRODUCT( (/2, 3, 4/) )
  ! The result is 24 because (2 * 3 * 4) = 24.

- **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  3  4 |
  !  4  5  6 |
  ! RES = PRODUCT(A)
  ! The result is 2880. All elements are multiplied.

  ! 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)

IBM Extension

Purpose
Convert to extended complex type.

Class
Elemental function

Argument type and attributes
- 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.

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).

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>QCMPLX</td>
<td>REAL(16)</td>
<td>COMPLEX(16)</td>
<td>no</td>
</tr>
</tbody>
</table>

Related information
"CMPLX(X, Y, KIND)" on page 496, "DCMPLX(X, Y)" on page 507.

End of IBM Extension

QEXT(A)

IBM Extension

Purpose
Convert to extended precision real type.
Class
Elemental function

Argument type and attributes
A must be of type integer, or real.

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.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>QFLOAT</td>
<td>any integer</td>
<td>REAL(16)</td>
<td>no</td>
</tr>
<tr>
<td>QEXT</td>
<td>default real</td>
<td>REAL(16)</td>
<td>no</td>
</tr>
<tr>
<td>QEXTD</td>
<td>double precision real</td>
<td>REAL(16)</td>
<td>no</td>
</tr>
</tbody>
</table>

RADIX(X)

Purpose
Returns the base of the model representing numbers of the same type and kind type parameter as the argument.

Class
Inquiry function

Argument type and attributes
X must be of type integer or real. It may be scalar or array valued.

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 475.
RAND()  

Purpose  
Not recommended. Generates uniform random numbers, positive real numbers greater than or equal to 0.0 and less than 1.0. Instead, use the standards conforming [RANDOM_NUMBER(HARVEST)] 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 594 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.

```fortran
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.

End of IBM Extension

RANDOM_NUMBER(HARVEST)

Purpose  
Returns one pseudo-random number or an array of pseudo-random numbers from the uniform distribution over the range 0 ≤ x < 1.

If you link the libthreads.a library, a parallel implementation of random number generation is employed which improves performance on SMP machines. The number of threads used can be controlled by the [intrinheads=num] run-time option.

Class  
Subroutine
Argument type and attributes

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 $0 \leq x < 1$.

Examples

```fortran
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)

Purpose

Restarts or queries the pseudo-random number generator used by RANDOM_NUMBER.

Class

Subroutine

Argument type and attributes

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 $\geq N$. 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 $\geq N$. It is an INTENT(OUT) argument. The seed for the current generator is transferred to it.

---

IBM Extension

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.

---

End of IBM Extension

---

IBM Extension

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. A valid seed must be a whole
number between 1.0 and 2147483647.0 (2.0**31-1) for Generator 1 and between 1.0 and 281474976710656.0 (2.0**48) for Generator 2.

Generator 1 uses the multiplicative congruential method, with
$$S(I+1) = (16807.0 \times S(I)) \mod (2.0**31-1)$$

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
$$S(I+1) = (44,485,709,377,909.0 \times S(I)) \mod (2.0**48)$$

and
$$X(I+1) = S(I+1) / (2.0**48)$$

Generator 2 cycles after (2**48) 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.

---

**Examples**

```fortran
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)**

**Purpose**

Returns the decimal exponent range in the model representing integer or real numbers with the same kind type parameter as the argument.

**Class**

Inquiry function

**Argument type and attributes**

X must be of type integer, real, or complex. It may be scalar or array valued.

**Result type and attributes**

Default integer scalar.
Result value

1. For an integer argument, the result is:
   \[
   \text{INT}( \log_{10}( \text{HUGE}(X) ) )
   \]
2. For a real or complex argument, the result is:
   \[
   \text{INT}( \min( \log_{10}( \text{HUGE}(X) ), -\log_{10}( \text{TINY}(X) ) ) )
   \]

IBM Extension

Thus:

<table>
<thead>
<tr>
<th>Type</th>
<th>RANGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer(1)</td>
<td>2</td>
</tr>
<tr>
<td>integer(2)</td>
<td>4</td>
</tr>
<tr>
<td>integer(4)</td>
<td>9</td>
</tr>
<tr>
<td>integer(8)</td>
<td>18</td>
</tr>
<tr>
<td>real(4), complex(4)</td>
<td>37</td>
</tr>
<tr>
<td>real(8), complex(8)</td>
<td>307</td>
</tr>
<tr>
<td>real(16), complex(16)</td>
<td>291</td>
</tr>
</tbody>
</table>

End of IBM Extension

Examples

IBM Extension

X is of type real(4):

\[
\text{HUGE}(X) = 0.34 \times 10^{39} \\
\text{TINY}(X) = 0.11 \times 10^{-37} \\
\text{RANGE}(X) = 37
\]

End of IBM Extension

See “Data representation models” on page 475.

REAL(A, KIND)

Purpose

Convert to real type.

Class

Elemental function

Argument type and attributes

A must be of type integer, real, or complex.

KIND (optional)

must be a scalar integer initialization expression.

Result type and attributes

- Real.
Case (i): If \( A \) is of type integer or real and \texttt{KIND} is present, the kind type parameter is that specified by \texttt{KIND}. If \( A \) is of type integer or real and \texttt{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 \texttt{KIND} is present, the kind type parameter is that specified by \texttt{KIND}. If \( A \) is of type complex and \texttt{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**

\texttt{REAL (-3)} has the value \(-3.0\). \texttt{REAL ((3.2, 2.1))} has the value \(3.2\).

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL</td>
<td>default integer</td>
<td>default real</td>
<td>no</td>
</tr>
<tr>
<td>FLOAT</td>
<td>any integer</td>
<td>default real</td>
<td>no</td>
</tr>
<tr>
<td>SNGL</td>
<td>double precision real</td>
<td>default real</td>
<td>no</td>
</tr>
<tr>
<td>SNGLQ</td>
<td>REAL(16)</td>
<td>default real</td>
<td>no 2</td>
</tr>
<tr>
<td>DREAL</td>
<td>double complex</td>
<td>double precision real</td>
<td>no 2</td>
</tr>
<tr>
<td>QREAL</td>
<td>COMPLEX(16)</td>
<td>REAL(16)</td>
<td>no 2</td>
</tr>
</tbody>
</table>

**Notes:**

1. IBM Extension: the ability to specify a nondefault integer argument.
2. IBM Extension: the inability to pass the name as an argument.

**REPEAT(STRING, NCOPIES)**

**Purpose**

Concatenate several copies of a string.

**Class**

Transformational function

**Argument type and attributes**

\texttt{STRING}

must be scalar and of type character.

\texttt{NCOPIES}

must be scalar and of type integer. Its value must not be negative.

**Result type and attributes**

Character scalar with a length equal to \texttt{NCOPIES} \* \texttt{LENGTH(STRING)}, with the same kind type parameter as \texttt{STRING}.

**Result value**

The value of the result is the concatenation of \texttt{NCOPIES} copies of \texttt{STRING}. 
Examples
REEPEAT ('H', 2) has the value 'HH'. REPEAT ('XYZ', 0) has the value of a zero-length string.

RESHAPE(SOURCE, SHAPE, PAD, ORDER)

Purpose
Constructs an array of a specified shape from the elements of a given array.

Class
Transformational function

Argument type and attributes

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)).

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 585.

RRSPACING(X)

**Purpose**
Returns the reciprocal of the relative spacing of the model numbers near the argument value.

**Class**
Elemental function

**Argument type and attributes**

X must be of type real.

**Result type and attributes**
Same as X.

**Result value**
The result is:

\[
\text{ABS(FRACTION(X))} \times \text{FLOAT(RADIX(X))}^{\text{DIGITS(X)}}
\]

**Examples**

<table>
<thead>
<tr>
<th>IBM Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>RRSPACING (-3.0) = 0.75 \times 2^{24}. See &quot;Real data model&quot; on page 477</td>
</tr>
</tbody>
</table>

RSHIFT(I, SHIFT)

**Purpose**
Performs a logical shift to the right.

**Class**
Elemental function

**Argument type and attributes**

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).

**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.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSHIFT</td>
<td>any integer</td>
<td>same as argument</td>
<td>yes</td>
</tr>
</tbody>
</table>

End of IBM Extension

---

**SCALE(X,I)**

**Purpose**
Returns the scaled value: \( X \times 2.0^I \)

**Class**
Elemental function

**Argument type and attributes**
- \( X \) must be of type real.
- \( I \) must be of type integer.

**Result type and attributes**
Same as \( X \).

**Result value**
The result is determined from the following:
\[
X \times 2.0^I
\]

\( SCALE (X, I) = X \times (2.0^I) \)

End of IBM Extension

---

Chapter 13. Intrinsic procedures 581
**Examples**

<table>
<thead>
<tr>
<th>IBM Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCALE (4.0, 3) = 4.0 * (2^3) = 32.0. See &quot;Real data model&quot; on page 477</td>
</tr>
</tbody>
</table>

---

**SCAN(STRING, SET, BACK)**

**Purpose**
Scan a string for any one of the characters in a set of characters.

**Class**
Elemental function

**Argument type and attributes**
- **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.

**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)**

**Purpose**
Returns a value of the kind type parameter of an integer data type that represents all integer values n with \(-10^R < n < 10^R\).
Class
Transformational function

Argument type and attributes
R must be a scalar of type integer.

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

---

IBM Extension

SELECTED_INT_KIND (9) has the value 4, signifying that an INTEGER with kind type 4 can represent all values from \(10^9\) to \(10^9\).

---

End of IBM Extension

Related information
Kind type parameters supported by XL Fortran are defined in "Type parameters and specifiers" on page 15.

SELECTED_REAL_KIND(P, R)

Purpose
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.

Class
Transformational function

Argument type and attributes

- \(P\) (optional)
  must be scalar and of type integer.

- \(R\) (optional)
  must be scalar and of type integer.

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

<table>
<thead>
<tr>
<th>IBM Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECTED_REAL_KIND (6, 70) has the value 8.</td>
</tr>
<tr>
<td>End of IBM Extension</td>
</tr>
</tbody>
</table>

Related information

Kind type parameters supported by XL Fortran are defined in “Type parameters and specifiers” on page 15.

SET_EXPONENT(X,I)

Purpose

Returns the number whose fractional part is the fractional part of the model representation of X, and whose exponent part is I.

Class

Elemental function

Argument type and attributes

- X must be of type real.
- I must be of type integer.

Result type and attributes

Same as X.

Result value

<table>
<thead>
<tr>
<th>IBM Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>If X = 0 the result is zero.</td>
</tr>
<tr>
<td>Otherwise, the result is:</td>
</tr>
<tr>
<td>FRACTION(X) * 2.0 I</td>
</tr>
<tr>
<td>End of IBM Extension</td>
</tr>
</tbody>
</table>
Examples

IBM Extension

SET_EXPONENT (10.5, 1) = 0.65625 * 2.0^1 = 1.3125

See “Real data model” on page 477.

End of IBM Extension

SHAPE(SOURCE)

Purpose
Returns the shape of an array or scalar.

Class
Inquiry function

Argument type and attributes
SOURCE is an array or scalar of any data type. It must not be a disassociated pointer, allocatable object that is not allocated, or assumed-size array.

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 579.

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)

Purpose
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 -qxl90=signedzero compiler option.
Class
Elemental function

Argument type and attributes

A must be of type integer or real.
B must be of the same type and kind type parameter as A.

Result type and attributes
Same as A.

Result value
The result is $sgn \times |A|$, where:
- $sgn = -1$, if either of the following is true:
  - $B < 0$

  [IBM Extension]
  - $B$ is a REAL(4) or REAL(8) number with a value of negative 0, and you have specified the -qxlf90=signedzero option
  [End of IBM Extension]

- $sgn = 1$, otherwise.

[Fortran 95]
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 f95 invocation commands.

[End of Fortran 95]

Examples
SIGN (-3.0, 2.0) has the value 3.0.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGN</td>
<td>default real</td>
<td>default real</td>
<td>yes</td>
</tr>
<tr>
<td>ISIGN</td>
<td>any integer</td>
<td>same as argument</td>
<td>yes</td>
</tr>
<tr>
<td>DSIGN</td>
<td>double precision real</td>
<td>double precision real</td>
<td>yes</td>
</tr>
<tr>
<td>QSIGN</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
<td>yes</td>
</tr>
</tbody>
</table>

Notes:
1. IBM Extension: the ability to specify a nondefault integer argument.
2. IBM Extension: the ability to pass the name as an argument.

Related information
See the -qxlf90 option in the XL Fortran Compiler Reference.
The SIGNAL procedure allows a program to specify a procedure to be invoked upon receipt of a specific operating-system signal.

**Class**
Subroutine

**Argument type and attributes**

- **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 fexc.p.h.

- **PROC** specifies the user-defined procedure to be invoked when the process receives the specified signal specified by argument I. It is an INTENT(IN) argument.

**Examples**

```fortran
INCLUDE 'fexc.p.h'
INTEGER SIGUSR1
EXTERNAL USRINT
! Set exception handler to produce the traceback code.
! The SIGTRAP is defined in the include file fexc.p.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**
The `-qsigtrap` option in the *XL Fortran Compiler Reference* allows you to set a handler for SIGTRAP signals through a compiler option.

---

**SIN(X)**

**Purpose**
Sine function.

**Class**
Elemental function
Argument type and attributes

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 and imaginary parts are regarded as values in radians.

Result type and attributes

Same as X.

Result value

It approximates sin(X).

Examples

SIN (1.0) has the value 0.84147098 (approximately).

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIN</td>
<td>default real</td>
<td>default real</td>
<td>yes</td>
</tr>
<tr>
<td>DSIN</td>
<td>double precision real</td>
<td>double precision real</td>
<td>yes</td>
</tr>
<tr>
<td>QSIN</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
<td>yes</td>
</tr>
<tr>
<td>CSIN</td>
<td>default complex</td>
<td>default complex</td>
<td>yes</td>
</tr>
<tr>
<td>CDSIN</td>
<td>double complex</td>
<td>double complex</td>
<td>yes</td>
</tr>
<tr>
<td>ZSIN</td>
<td>double complex</td>
<td>double complex</td>
<td>yes</td>
</tr>
<tr>
<td>CQSIN</td>
<td>COMPLEX(16)</td>
<td>COMPLEX(16)</td>
<td>yes</td>
</tr>
</tbody>
</table>

Notes:

1. IBM Extension: the ability to pass the name as an argument.
2. Given that X is a complex number in the form a + bi, where i = (-1)½:
   a. abs(b) must be less than or equal to 88.7228; a is any real value.
   b. abs(b) must be less than or equal to 709.7827; a is any real value.

SIND(X)

Purpose

Sine function. Argument in degrees.

Class

Elemental function

Argument type and attributes

X must be of type real.

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.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIND</td>
<td>default real</td>
<td>default real</td>
<td>yes</td>
</tr>
<tr>
<td>DSIND</td>
<td>double precision real</td>
<td>double precision real</td>
<td>yes</td>
</tr>
<tr>
<td>QSIND</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
<td>yes</td>
</tr>
</tbody>
</table>

End of IBM Extension

SINH(X)

Purpose
Hyperbolic sine function.

Class
Elemental function

Argument type and attributes
X must be of type real.

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).

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>SINH 1</td>
<td>default real</td>
<td>default real</td>
<td>yes</td>
</tr>
<tr>
<td>DSINH 2</td>
<td>double precision real</td>
<td>double precision real</td>
<td>yes</td>
</tr>
<tr>
<td>QSINH 2 3</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
<td>yes</td>
</tr>
</tbody>
</table>

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.
3. IBM Extension.

SIZE(ARRAY, DIM)

Purpose
Returns the extent of an array along a specified dimension or the total number of elements in the array.

Class
Inquiry function
**Argument type and attributes**

**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 $1 \leq \text{DIM} \leq \text{rank(ARRAY)}$.

---

**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**

```fortran
! 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.
```

---

**SIZEOF(A)**

---

**IBM Extension**

---

**Purpose**

Returns the size of an argument in bytes.

---

**Class**

Inquiry function

---

**Argument type and attributes**

**A**

is a data object that cannot be any of the following:

- A Fortran 90 pointer
- An automatic object
- An allocatable object
- A derived object or record structure that has an allocatable or a Fortran 90 pointer component
- An array section
- An array constructor
- An assumed-shape array
- A whole assumed-size array
• A zero-sized array
• A derived object or record structure containing components not accessible within the scoping unit

`SIZEOF` must not be passed as an argument to a subprogram.

**Result type and attributes**
Default integer scalar.

**Result value**
The size of the argument in bytes.

**Examples**
The following example assumes that `-qintsize=4`.

```fortran
INTEGER ARRAY(10)
INTEGER*8, PARAMETER :: p = 8
STRUCTURE /STR/
  INTEGER I
  COMPLEX C
END STRUCTURE
RECORD /STR/ R
CHARACTER*10 C
TYPE DTYPE
  INTEGER ARRAY(10)
END TYPE
TYPE (DTYPE) DOBJ
PRINT *, SIZEOF(ARRAY), SIZEOF (ARRAY(3)), SIZEOF(P) ! Array, array 
  ! element ref, 
  ! named constant

PRINT *, SIZEOF (R), SIZEOF(R.C) ! record structure 
  ! entity, record 
  ! structure 
  ! component

PRINT *, SIZEOF (C(2:5)), SIZEOF(C) ! character 
  ! substring, 
  ! character 
  ! variable

PRINT *, SIZEOF (DOBJ), SIZEOF(DOBJ%ARRAY) ! derived type 
  ! object, structure 
  ! component
```

The following is sample output generated by the program above:

```
  40  4  8
 16   8
  4 10
  40 40
```

**Related information**
See the [XL Fortran Compiler Reference](#) for details about the `qintsize` compiler option.

[End of IBM Extension]
**SPACING(X)**

**Purpose**
Returns the absolute spacing of the model numbers near the argument value.

**Class**
Elemental function

**Argument type and attributes**
X must be of type real.

**Result type and attributes**
Same as X.

**Result value**
If X is not 0, the result is:

\[ 2.0^{\text{EXponent}(X) - \text{DIGITS}(X)} \]

If X is 0, the result is the same as that of TINY(X).

**Examples**

<table>
<thead>
<tr>
<th>IBM Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPACING (3.0) = 2.0^{2} \cdot 2^{24} = 2.0^{(22)} See “Real data model” on page 477.</td>
</tr>
</tbody>
</table>

**End of IBM Extension**

---

**SPREAD(SOURCE, DIM, NCOPIES)**

**Purpose**
Replicates an array in an additional dimension by making copies of existing elements along that dimension.

**Class**
Transformational function

**Argument type and attributes**

- **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.
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)

Purpose
Square root.

Class
Elemental function

Argument type and attributes
X must be of type real or complex. Unless X is complex, its value must be greater than or equal to zero.

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.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQRT</td>
<td>default real</td>
<td>default real</td>
<td>yes</td>
</tr>
<tr>
<td>DSQRT</td>
<td>double precision</td>
<td>double precision</td>
<td>yes</td>
</tr>
<tr>
<td>QSQR</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
<td>yes 1</td>
</tr>
<tr>
<td>CSQRT</td>
<td>default complex</td>
<td>default complex</td>
<td>yes</td>
</tr>
<tr>
<td>CDSQRT</td>
<td>double complex</td>
<td>double complex</td>
<td>yes 1</td>
</tr>
<tr>
<td>ZSQRT</td>
<td>COMPLEX(8)</td>
<td>COMPLEX(8)</td>
<td>yes 1</td>
</tr>
<tr>
<td>CQSQRT</td>
<td>COMPLEX(16)</td>
<td>COMPLEX(16)</td>
<td>yes 1</td>
</tr>
</tbody>
</table>

Notes:
1. IBM Extension: the ability to pass the name as an argument.
2. Given that X is a complex number in the form a + bi, where i = (-1)½, abs(X) + abs(a) must be less than or equal to 1.797693 * 10^308.

SRAND(SEED)

IBM Extension

Purpose

Provides the seed value used by the random number generator function RAND.

This intrinsic subroutine is not recommended. Use the standards conforming RANDOM_NUMBER(HARVEST) intrinsic subroutine.

Class

Subroutine

Argument type and attributes

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.

Examples

The following is an example of a program using the SRAND subroutine.

```fortran
CALL SRAND(0.5)
DO I = 1, 5
   R = RAND()
   PRINT *,R
ENDDO
END
```

The following is sample output generated by the above program:
SUM(ARRAY, DIM, MASK) or SUM(ARRAY, MASK)

Purpose
Calculates the sum of selected elements in an array.

Class
Transformational function

Argument type and attributes

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRAY</td>
<td>is an array of numeric type, whose elements you want to sum.</td>
</tr>
<tr>
<td>DIM (optional)</td>
<td>is an integer scalar in the range 1 ≤ DIM ≤ rank(ARRAY).</td>
</tr>
<tr>
<td>MASK (optional)</td>
<td>is a logical expression. If it is an array, it must conform with</td>
</tr>
<tr>
<td></td>
<td>ARRAY in shape. If MASK is a scalar, the scalar value applies to all</td>
</tr>
<tr>
<td></td>
<td>elements in ARRAY.</td>
</tr>
</tbody>
</table>

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.

Fortran 95
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

End of Fortran 95

**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 |
  | (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 |
  | (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 |
  | (4 + 3) = 7
  | (7 + 8 + 5) = 20
```

**SYSTEM(CMD, RESULT)**

**IBM Extension**

**Purpose**

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.

**Class**

Subroutine

**Argument type and attributes**

- **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.

Examples

```plaintext
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

```

End of IBM Extension

### System_Clock(COUNT, COUNT_RATE, COUNT_MAX)

#### Purpose

Returns integer data from a real-time clock.

#### Class

Subroutine

#### Argument type and attributes

- **COUNT (optional)** is an INTENT(OUT) argument that must be scalar and of type default integer. The initial value of COUNT depends on the current value of the processor clock in a range from 0 to COUNT_MAX. COUNT increments by one for each clock count until it reaches the value of COUNT_MAX. At the next clock count after COUNT_MAX, the value of COUNT resets to zero.

- **COUNT_RATE (optional)** is an INTENT(OUT) argument that must be scalar and of type default integer. When using the default centisecond resolution, COUNT_RATE refers to the number of processor clock counts per second or to zero if there is no clock.

  If you specify a microsecond resolution using `-qsclk=micro`, the value of COUNT_RATE is 1 000 000 clock counts per second.

- **COUNT_MAX (optional)** is an INTENT(OUT) argument that must be scalar
and of type default integer. When using the default centisecond resolution, COUNT_MAX is the maximum number of clock counts for a given processor clock.

If you specify a microsecond resolution using -qsclk=micro and a default of INTEGER(4), the value of COUNT_MAX is 1 799 999 999 clock counts, or about 30 minutes.

If you specify a microsecond resolution using -qsclk=micro and a default of INTEGER(8), the value of COUNT_MAX is 86 399 999 999 clock counts, or about 24 hours.

Examples

IBM Extension

In the following example, the 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 implementation dependent.

```fortran
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
```

End of IBM Extension

Related information

See the -qsclk compiler option in the XL Fortran Compiler Reference for more information on specifying system clock resolution.

TAN(X)

Purpose

Tangent function.
Class
Elemental function

**Argument type and attributes**

\(X\) must be of type real.

**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).

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAN</td>
<td>default real</td>
<td>default real</td>
<td>yes</td>
</tr>
<tr>
<td>DTAN</td>
<td>double precision real</td>
<td>double precision real</td>
<td>yes</td>
</tr>
<tr>
<td>QTAN</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Notes:**

1. IBM Extension: the ability to pass the name as an argument.

---

**TAND(X)**

---

**IBM Extension**

**Purpose**

Tangent function. Argument in degrees.

**Class**

Elemental function

**Argument type and attributes**

\(X\) must be of type real.

**Result type and attributes**

Same as \(X\).

**Result value**

The result approximates \(\tan(X)\), where \(X\) has a value in degrees.

**Examples**

\(\tan(45.0)\) has the value 1.0.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAND</td>
<td>default real</td>
<td>default real</td>
<td>yes</td>
</tr>
<tr>
<td>DTAND</td>
<td>double precision real</td>
<td>double precision real</td>
<td>yes</td>
</tr>
</tbody>
</table>
### TANH(X)

**Purpose**
Hyperbolic tangent function.

**Class**
Elemental function

**Argument type and attributes**
- $X$ must be of type real.

**Result type and attributes**
Same as $X$.

**Result value**
The result has a value equal to tanh($X$).

**Examples**
- $\text{TANH}(1.0)$ has the value 0.76159416 (approximately).

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
<th>Pass As Arg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>TANH</td>
<td>default real</td>
<td>default real</td>
<td>yes</td>
</tr>
<tr>
<td>DTANH</td>
<td>double precision real</td>
<td>double precision real</td>
<td>yes</td>
</tr>
<tr>
<td>QTANH</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Notes:**
1. IBM Extension: the ability to pass the name as an argument.

### TINY(X)

**Purpose**
Returns the smallest positive number in the model representing numbers of the same type and kind type parameter as the argument.

**Class**
Inquiry function

**Argument type and attributes**
- $X$ must be of type real. It may be scalar or array valued.

**Result type and attributes**
Scalar with the same type as $X$. 
**Result value**

IBM Extension

The result is:

\[ 2.0^{\text{MINEXPONENT}(X) - 1} \quad \text{for real } X \]

End of IBM Extension

**Examples**

IBM Extension

\[ \text{TINY}(X) = \text{float}(2)^{(-126)} = 1.17549351e-38. \]

See “Real data model” on page 477.

End of IBM Extension

**TRANSFER(SOURCE, MOLD, SIZE)**

**Purpose**

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.

**Class**

Transformational function

**Argument type and attributes**

**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. If MOLD is a variable, the value does not need to be defined. 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.

**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:

```fortran
REAL(4) X /3.141/
DOUBLE PRECISION 1, 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)**

**Purpose**

Transposes a two-dimensional array, turning each column into a row and each row into a column.

**Class**

Transformational function

**Argument type and attributes**

- **MATRIX** is an array of any data type, with a rank of two.

**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.

**Result type and attributes**

A two-dimensional array of the same data type as MATRIX.
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)**

**Purpose**

Returns the argument with trailing blank characters removed.

**Class**

Transformational function

**Argument type and attributes**

- **STRING**: must be of type character and must be a scalar.

**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 ('bAbBbb') has the value 'bAbB'.
```

**UBOUND(ARRAY, DIM)**

**Purpose**

Returns the upper bounds of each dimension in an array, or the upper bound of a specified dimension.

**Class**

Inquiry function

**Argument type and attributes**

- **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 $1 \leq \text{DIM} \leq \text{rank(ARRAY)}$. The corresponding actual argument must not be an optional dummy argument.

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(:,:,1) )
! 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)

Purpose

Takes some or all elements from a one-dimensional array and rearranges them into another, possibly larger, array.

Class

Transformational function

Argument type and attributes

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..

Result value
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.

Result type and attributes
An array with the same shape as MASK and the same data type as VECTOR.

Examples

```fortran
! VECTOR is the array (/ 5, 6, 7, 8 /),
! MASK is [ F T T ], FIELD is [ -1 -4 -7 ]
! [ ] F F F
! [ ] F T T [ -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
```

VERIFICATION(STRING, SET, BACK)

Purpose
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.

Class
Elemental function

Argument type and attributes

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRING</td>
<td>must be of type character.</td>
<td></td>
</tr>
<tr>
<td>SET</td>
<td>must be of type character</td>
<td>with the same kind type parameter as STRING.</td>
</tr>
<tr>
<td>BACK (optional)</td>
<td>must be of type logical.</td>
<td></td>
</tr>
</tbody>
</table>

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 14. Hardware-specific intrinsic procedures

This section provides an alphabetical reference to the hardware-specific intrinsic functions. Many of these intrinsics provide access to hardware instructions that may not strictly conform to all IEEE floating-point semantic rules depending on their usage. You should exercise caution if strict IEEE floating-point conformance is important to your application.

FCFI(I)

**Purpose**
Floating-point Conversion from Integer

Converts an integer value in a floating-point variable into a floating-point value.

This intrinsic is valid on any 64-bit PowerPC architecture.

**Class**
Function

**Argument type and attributes**

I must be of type REAL(8).

**Result type and attributes**

Same as I.

**Result value**

The double-precision floating-point value of I.

**Examples**

```fortran
... REAL*8 :: R8, RES INTEGER*8 :: I8 EQUIVALENCE(R8, I8)
I8 = 89
RES = FCFI(R8) ! RES = 89.0
...
```

FCTID(X)

**Purpose**
Floating-point Convert to Integer

Converts a floating-point operand into a 64-bit, signed fixed-point integer using the current rounding mode.
This intrinsic is valid on any 64-bit PowerPC architecture.

**Class**
Function

**Argument type and attributes**

\[ X \quad \text{must be of type} \quad \text{REAL(8)}. \]

**Result type and attributes**

Same as \( X \).

**Result value**

The result is a fixed-point integer, inside a floating-point register.

---

**FCTIDZ(X)**

**Purpose**

Floating-point Convert to Integer Round to Zero

Converts a floating-point operand into a 64-bit signed fixed-point integer and rounds to zero.

This intrinsic is valid on any 64-bit PowerPC architecture.

**Class**
Function

**Argument type and attributes**

\[ X \quad \text{must be of type} \quad \text{REAL(8)}. \]

**Result type and attributes**

Same as \( X \).

**Result value**

The result is a fixed-point integer, inside a floating-point variable, rounded to zero.

---

**FCTIW(X)**

**Purpose**

Floating-point Convert to Integer

Converts a floating-point operand into a 32-bit, signed fixed-point integer using the current rounding mode.

**Class**
Function
**Argument type and attributes**

X must be of type REAL(8).

**Result type and attributes**

Same as X.

**Result value**

The result is a fixed-point integer, inside a floating point variable.

---

**FCTIWZ(X)**

**Purpose**

Floating-point Convert to Integer Round to Zero

Converts a floating-point operand into a 32-bit signed fixed-point integer and rounds to zero.

**Class**

Function

**Argument type and attributes**

X must be of type REAL(8).

**Result type and attributes**

Same as X.

**Result value**

The result is a fixed-point integer, inside a floating point variable, rounded to zero.

---

**FMADD(A, X, Y)**

**Purpose**

Floating-point Multiply and Add

Returns the result of a floating-point multiply-add.

**Class**

Function

**Argument type and attributes**

A can be of type REAL(4) or REAL(8).

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.

**Result type and attributes**

Same as A, X, and Y.
Result value

The result has a value equal to $A \times X + Y$.

Examples

The following example is only valid if compiled with the -qarch option 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
```

FMSUB(A, X, Y)

Purpose

Floating-point Multiply and Subtract

Returns the result of a floating-point multiply–subtract.

Class

Function

Argument type and attributes

- $A$ can be of type REAL(4) or REAL(8).
- $X$ must be of the same type and kind type parameter as $A$.
- $Y$ must be of the same type and kind type parameter as $A$.

Result type and attributes

Same as $A$, $X$, and $Y$.

Result value

The result has a value equal to $A \times X - Y$.

FNABS(X)

Purpose

Returns the negative floating-point value $-|X|$.

Class

Function

Argument type and attributes

- $X$ must be of type REAL.

Result type and attributes

Same as $X$. 
Result value

The result is a negative floating-point value of \( X, -|X| \).

Examples

In the following example, the absolute content of a floating-point variable is negated.

```fortran
REAL(4) :: A, RES1
REAL(8) :: D, RES2
RES1 = FNABS(A)
RES2 = FNABS(D)
```

**FNMADD(A, X, Y)**

**Purpose**

Floating-point Negative Multiply and Add

Returns the result of a floating-point negative multiply-add.

**Class**

Function

**Argument type and attributes**

\( A \) can be of type \( \text{REAL}(4) \) or \( \text{REAL}(8) \).

\( X \) must be of the same type and kind type parameter as \( A \).

\( Y \) must be of the same type and kind type parameter as \( A \).

**Result type and attributes**

Same as \( X \).

**Result value**

The result has a value equal to \(- (A \times X + Y)\).

**FNMSUB(A, X, Y)**

**Purpose**

Floating-point Negative Multiply and Subtract

Returns the result of a floating-point negative multiply–subtract.

**Class**

Function

**Argument type and attributes**

\( A \) can be of type \( \text{REAL}(4) \) or \( \text{REAL}(8) \).

\( X \) must be of the same type and kind type parameter as \( A \).

\( Y \) must be of the same type and kind type parameter as \( A \).
**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 \texttt{FNMSUB} is of type \texttt{REAL(4)}. It is converted to \texttt{REAL(8)} and then assigned to \texttt{RES}.

\begin{verbatim}
REAL(4) :: A, B, C
REAL(8) :: RES
RES = FNMSUB(A, B, C)
END
\end{verbatim}

**FRE(X)**

**Purpose**
Floating-point Reciprocal Estimate

Returns an estimate of a floating-point reciprocal operation.

Valid on a POWER5 architecture.

**Class**
Function

**Argument type and attributes**
\( X \) must be of type \texttt{REAL(8)}. 

**Result type and attributes**
Same as \( X.\)

**Result value**
The result is a double precision estimate of \( 1/X.\)

**FRES(X)**

**Purpose**
Floating-point Reciprocal Estimate Single

Returns an estimate of a floating-point reciprocal operation.

Valid on any PowerPC with extended graphics opcodes.

**Class**
Function
Argument type and attributes
   \( X \) must be of type \( \text{REAL}(4) \).

Result type and attributes
   Same as \( X \).

Result value
   The result is a single precision estimate of \( 1/X \).

---

**FRIM(A)**

Purpose
   Floating-point Round to Integer Minus
   Valid on a POWER5+ architecture.

Class
   Function

Argument type and attributes
   \( A \) must be of type \( \text{REAL}(4) \) or \( \text{REAL}(8) \).

Result type and attributes
   Same as \( A \).

Result value
   The result has a value equal to the greatest integer less than or equal to \( A \).

---

**FRIN(A)**

Purpose
   Floating-point Round to Integer Nearest
   Valid on a POWER5+ architecture.

Class
   Function

Argument type and attributes
   \( A \) must be of type \( \text{REAL}(4) \) or \( \text{REAL}(8) \).

Result type and attributes
   Same as \( A \).
Result value
If \( A > 0 \), \( \text{FRIN}(A) \) has the value \( \text{FRIM}(A + 0.5) \).
If \( A \leq 0 \), \( \text{FRIN}(A) \) has the value \( \text{FRIM}(A - 0.5) \).

\text{FRIP}(A)

Purpose
Floating-point Round to Integer Plus
Valid on a POWER5+ architecture.

Class
Function

Argument type and attributes
\( A \) must be of type REAL(4) or REAL(8).

Result type and attributes
Same as \( A \).

Result value
The result has a value equal to the least integer greater than or equal to \( A \).

\text{FRIZ}(A)

Purpose
Floating-point Round to Integer Zero
Valid on a POWER5 architecture.

Class
Function

Argument type and attributes
\( A \) must be of type REAL(4) or REAL(8).

Result type and attributes
Same as \( A \).

Result value
If \( A > 0 \), \( \text{FRIZ}(A) \) has the value \( \text{FRIM}(A) \).
If \( A \leq 0 \), \( \text{FRIZ}(A) \) has the value \( \text{FRIP}(A) \).
FRSQRTE(X)

**Purpose**
Floating-point Square Root Reciprocal Estimate
Returns the result of a reciprocal square root operation.
Valid on any PowerPC with extended graphics opcodes.

**Class**
Function

**Argument type and attributes**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type/Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>REAL(8)</td>
</tr>
</tbody>
</table>

**Result type and attributes**
Same as X.

**Result value**
The result is a double precision estimate of the reciprocal of the square root of X.

FRSQRTES(X)

**Purpose**
Floating-point Square Root Reciprocal Estimate Single
Returns the result of a reciprocal square root operation.
Valid on a POWER5 architecture.

**Class**
Function

**Argument type and attributes**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type/Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>REAL(4)</td>
</tr>
</tbody>
</table>

**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.

FSEL(X,Y,Z)

**Purpose**
Floating-point Selection
Returns the result of a floating-point selection operation. This result is determined by comparing the value of \( X \) with zero.

Valid on any PowerPC with extended graphics opcodes.

**Class**

Function

**Argument type and attributes**

\( X \) must be of type REAL(4) or REAL(8).

**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.

---

**MTFSF(MASK, R)**

**Purpose**

Move to floating-point status and control register (FPSCR) fields

The contents of \( R \) are placed into the FPSCR under control of the field mask specified in \( MASK \).

**Class**

Subroutine

**Argument type and attributes**

\( MASK \) must be a literal value of type INTEGER(4). The lower eight bits are used.

\( R \) must be of type REAL(8).

---

**MTFSI(BF, I)**

**Purpose**

Move to floating-point status and control register (FPSCR) Fields Immediate

The value of \( I \) is placed into FPSCR field specified in \( BF \).

**Class**

Subroutine
Argument type and attributes

- BF must be a literal value from 0 to 7, of type INTEGER(4).
- I must be a literal value from 0 to 15, of type INTEGER(4).

MULHY(RA, RB)

Purpose
Returns the high-order 32 or 64–bits of the 64-bit or 128-bit products of the operands RA and RB.

Valid on a PowerPC with a 64-bit architecture, in 64–bit mode for 64–bit integers.

Valid for 64–bit integers in 64–bit mode.

Class
Function

Argument type and attributes

- RA must be of type integer.
- RB must be of type integer.

Result type and attributes
Same as RA, RB.

Result value
A 32 or 64–bit product of the operands RA and RB

POPCNTB(I)

IBM Extension

Purpose
Population count.

Counts the number of set bits of each byte in a register.

Valid only on POWER5.

Class
Elemental function.

Argument type and attributes

- I An INTENT(IN) argument of type INTEGER(4) in 32–bit mode.
  An INTENT(IN) argument of type INTEGER(4) or INTEGER(8) in 64–bit mode
**Result type and attributes**

Returns an INTEGER(4) in 32–bit mode.

Returns an INTEGER(8) in 64–bit mode.

**Result value**

The number of bits set to on in that byte, in the position of the byte.

**Examples**

```fortran
INTEGER I
I = x'01030bff'
WRITE(*, ' (Z8.8)') POPCNTB(I)
END
```

Expected output:

01020008

**Related information**

- Data representation models

---

**ROTATELI(RS, IS, SHIFT, MASK)**

**Purpose**

Rotate Left Immediate then MASK Insert

Rotates the value of RS left by the number of bits specified in SHIFT. The function then inserts RS into IS under bit mask, MASK.

**Class**

Function

**Argument type and attributes**

- **RS** must be of type integer.
- **IS** must be of type integer.
- **SHIFT** must be a literal value. For 4-byte RS values, the SHIFT value will be truncated to the last five bits. For 8-byte RS values, the SHIFT value will be truncated to the last six bits.
- **MASK** must be a literal value of type integer.

**Result type and attributes**

Same as **RS**.

**Result value**

Rotates **RS** left the number of bits specified by **SHIFT**, and inserts the result into **IS** under the bit mask, **MASK**.
ROTATELM(RS, SHIFT, MASK)

**Purpose**

Rotate Left AND with Mask

Rotates the value of RS left by the number of bits specified in SHIFT. The rotated data is ANDed with the MASK and then returned as a result.

**Class**

Function

**Argument type and attributes**

- **RS** must be of type integer.
- **SHIFT** must be a literal value. For 4-byte RS values, the SHIFT value will be truncated to the last five bits. For 8-byte RS values, the SHIFT value will be truncated to the last six bits.
- **MASK** must be a literal value of type integer.

**Result type and attributes**

Same as RS.

**Result value**

The rotated data ANDed with MASK.

SETFSB0(BT)

**Purpose**

Move 0 to floating-point status and control register (FPSCR) bit.

Bit BT of FPSCR is set to 0. This subroutine returns no value.

**Class**

Subroutine

**Argument type and attributes**

- **BT** must be of type INTEGER(4) and a literal value.

SETFSB1(BT)

**Purpose**

Move 1 to FPSCR bit.

Bit BT of FPSCR is set to 1. This subroutine returns no value.
Valid on any PowerPC.

**Class**
Subroutine

**Argument type and attributes**

- **BT** must be of type `INTEGER(4)` and a literal value.

---

### SFTI(M, Y)

**Purpose**

Store Floating-point to Integer

The contents of the low order 32-bits of `Y` are stored without conversion into `M`.

Valid on any PowerPC.

**Class**
Subroutine

**Argument type and attributes**

- **M** must be of type `INTEGER(4)`.
- **Y** must be of type `REAL(8)`.

**Examples**

```fortran
integer*4 :: m
real*8 :: x

x = z"00000000abcd0001"
call sfti(m, x) ! m = z"abcd0001"
```

---

### SWDIV(X,Y)

**Purpose**

Provides software floating-point division algorithms when targeting and running on POWER5 processors.

This function returns the result of a floating-point division and can increase performance over the normal divide operator where your application performs division repeatedly within a loop.

**Class**
Elemental function

**Argument type and attributes**

- **X** can be of type `REAL(4)` or `REAL(8)`.
- **Y** must be of the same type and kind type parameter as `X`. 
Result type and attributes

Same as X and Y.

Result value

The result has a value equal to X/Y.

For REAL(4) arguments, the result is bitwise identical to IEEE division.

For REAL(8) arguments with -qstrict in effect, the result is bitwise identical to IEEE division.

For REAL(8) arguments with -qnostrict in effect, the result can differ slightly from the IEEE result.

Examples

The following example uses software division algorithms if compiled with the -qarch=pwr5 option and run on a POWER5 processor.

```
REAL(4) :: A, B DIVRES1
REAL(8) :: E, F DIVRES2

DIVRES1 = SWDIV(A, B)
DIVRES2 = SWDIV(E, F)
END
```

**SWDIV_NOCHK(X,Y)**

**Purpose**

Provides software floating-point division algorithms when targeting and running on POWER5 processors. Checking for invalid arguments is not performed.

This function returns the result of a floating-point division and can increase performance over the normal divide operator or the SWDIV built-in function where your application performs division repeatedly within a loop, and arguments are within the permitted range.

**Class**

Elemental function

**Argument type and attributes**

X can be of type REAL(4) or REAL(8).

For a REAL(4) argument, you must not specify the following:

- \[ |\text{numerator}| \text{equal to infinity} \]
- \[ |\text{denominator}| \text{equal to infinity} \]
- \[ |\text{denominator}| < 2^{-1022} \]
- \[ |\text{numerator/denominator}| \text{equal to infinity} \]

For correct operation, REAL(8) arguments must satisfy the following conditions:

- \[ 2^{-970} < |\text{numerator}| < \text{Inf} \]
- \[ 2^{-1022} \leq |\text{denominator}| < 2^{1021} \]
\[
2^{(-1021)} < |\text{numerator/denominator}| < 2^{1023}
\]

\(Y\) must be of the same type and kind type parameter as \(X\).

**Result type and attributes**

Same as \(X\) and \(Y\).

**Result value**

The result has a value equal to \(X/Y\).

For \(\text{REAL}(4)\) arguments, the result is bitwise identical to IEEE division.

For \(\text{REAL}(8)\) arguments with -qstrict in effect, the result is bitwise identical to IEEE division.

For \(\text{REAL}(8)\) arguments with -qnostrict in effect, the result can differ slightly from the IEEE result.

---

**TRAP(A, B, TO)**

**Purpose**

Operand \(A\) is compared with operand \(B\). This comparison results in five conditions which are ANDed with \(TO\). If the result is not 0, the system trap handler is invoked.

8-byte integers are valid only in 64-bit mode.

Both operands \(A\) and \(B\) must be either of type \(\text{INTEGER}(4)\) or \(\text{INTEGER}(8)\).

**Class**

Subroutine

**Argument type and attributes**

\(A\) must be of type integer.

\(B\) must be of type integer.

\(TO\) must be a literal value from 1 to 31, of type \(\text{INTEGER}(4)\).

\[\text{------------------- End of IBM Extension -------------------}\]
Chapter 15. VMX intrinsic procedures

Individual elements of vectors can be accessed by using storage association, the TRANSFER intrinsic, or the Vector Multimedia eXtension (VMX) intrinsic functions. This section provides an alphabetical reference to the Vector Multimedia eXtension (VMX) intrinsic functions. These intrinsics allow you to manipulate vectors.

VEC_ABS(ARG1)

Purpose
Returns a vector containing the absolute values of the contents of the given vector.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) integer vector or real vector.

Result type and attributes
The result is of the same type as ARG1.

Result value
The value of each element of the result is the absolute value of the corresponding element of ARG1. For integer vectors, the arithmetic is modular.

VEC_ABSS(ARG1)

Purpose
Returns a vector containing the saturated absolute values of the contents of the given vector.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) integer vector.

Result type and attributes
The result is of the same type as ARG1.
Result value
The value of each element of the result is the saturated absolute value of the corresponding element of ARG1.

VEC_ADD(ARG1, ARG2)

Purpose
Returns a vector containing the sums of each set of corresponding elements of the given vectors.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) integer vector, unsigned vector, or real vector.
ARG2
An INTENT(IN) vector of the same type as ARG1.

Result type and attributes
The result is of the same type as ARG1.

Result value
The value of each element of the result is the sum of the corresponding elements of ARG1 and ARG2. For integer vectors and unsigned vectors, the arithmetic is modular.

VEC_ADDC(ARG1, ARG2)

Purpose
Returns a vector containing the carry produced by adding each set of corresponding elements of the given vectors.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) UNSIGNED(4) vector.
ARG2
An INTENT(IN) vector of the same type as ARG1.

Result type and attributes
The result is of the same type as ARG1.
Result value
The value of each element of the result is the carry produced by adding the corresponding elements of ARG1 and ARG2 (1 if there is a carry, 0 otherwise).

VEC_ADDS(ARG1, ARG2)

Purpose
Returns a vector containing the saturated sums of each set of corresponding elements of the given vectors.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) integer vector or unsigned vector.
ARG2
An INTENT(IN) vector of the same type as ARG1.

Result type and attributes
The result is of the same type as ARG1.

Result value
The value of each element of the result is the saturated sum of the corresponding elements of ARG1 and ARG2.

VEC_ALL_EQ(ARG1, ARG2)

Purpose
Tests whether all sets of corresponding elements of the given vectors are equal.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) vector.
ARG2
An INTENT(IN) vector of the same type as ARG1.

Result type and attributes
The result is an INTEGER(4).

Result value
The result is 1 if each element of ARG1 is equal to the corresponding element of ARG2. Otherwise, the result is 0.
VEC_ALL_GE(ARG1, ARG2)

Purpose
Tests whether all elements of the first argument are greater than or equal to the corresponding elements of the second argument.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) integer vector, real vector, or unsigned vector.
ARG2
An INTENT(IN) vector of the same type as ARG1.

Result type and attributes
The result is an INTEGER(4).

Result value
The result is 1 if all elements of ARG1 are greater than or equal to the corresponding elements of ARG2. Otherwise, the result is 0.

VEC_ALL_GT(ARG1, ARG2)

Purpose
Tests whether all elements of the first argument are greater than the corresponding elements of the second argument.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) integer vector, real vector, or unsigned vector.
ARG2
An INTENT(IN) vector of the same type as ARG1.

Result type and attributes
The result is an INTEGER(4).

Result value
The result is 1 if all elements of ARG1 are greater than the corresponding elements of ARG2. Otherwise, the result is 0.
VEC_ALL_IN(ARG1, ARG2)

**Purpose**
Tests whether each element of a given vector is within a given range.

**Class**
Elemental function

**Argument type and attributes**
ARG1
An INTENT(IN) real vector.
ARG2
An INTENT(IN) vector of the same type as ARG1.

**Result type and attributes**
The result is an INTEGER(4).

**Result value**
The result is 1 if all elements of ARG1 have values less than or equal to the value of the corresponding element of ARG2, and greater than or equal to the negative of the value of the corresponding element of ARG2. Otherwise, the result is 0.

VEC_ALL_LE(ARG1, ARG2)

**Purpose**
Tests whether all elements of the first argument are less than or equal to the corresponding elements of the second argument.

**Class**
Elemental function

**Argument type and attributes**
ARG1
An INTENT(IN) integer vector, real vector, or unsigned vector.
ARG2
An INTENT(IN) vector of the same type as ARG1.

**Result type and attributes**
The result is an INTEGER(4).

**Result value**
The result is 1 if all elements of ARG1 are less than or equal to the corresponding elements of ARG2. Otherwise, the result is 0.
VEC_ALL_LT(ARG1, ARG2)

**Purpose**
Tests whether all elements of the first argument are less than the corresponding elements of the second argument.

**Class**
Elemental function

**Argument type and attributes**
- **ARG1**
  An INTENT(IN) integer vector, real vector, or unsigned vector.
- **ARG2**
  An INTENT(IN) vector of the same type as ARG1.

**Result type and attributes**
The result is an INTEGER(4).

**Result value**
The result is 1 if all elements of ARG1 are less than the corresponding elements of ARG2. Otherwise, the result is 0.

VEC_ALL_NAN(ARG1)

**Purpose**
Tests whether each element of the given vector is a NaN.

**Class**
Elemental function

**Argument type and attributes**
- **ARG1**
  An INTENT(IN) real vector.

**Result type and attributes**
The result is an INTEGER(4).

**Result value**
The result is 1 if each element of ARG1 is a NaN. Otherwise, the result is 0.

VEC_ALL_NE(ARG1, ARG2)

**Purpose**
Tests whether all sets of corresponding elements of the given vectors are not equal.

**Class**
Elemental function
**Argument type and attributes**

**ARG1**
An INTENT(IN) vector.

**ARG2**
An INTENT(IN) vector of the same type as ARG1.

**Result type and attributes**
The result is an INTEGER(4).

**Result value**
The result is 1 if each element of ARG1 is not equal to the corresponding element of ARG2. Otherwise, the result is 0.

---

**VEC_ALL_NGE(ARG1, ARG2)**

**Purpose**
Tests whether each element of the first argument is not greater than or equal to the corresponding element of the second argument.

**Class**
Elemental function

**Argument type and attributes**

**ARG1**
An INTENT(IN) real vector.

**ARG2**
An INTENT(IN) vector of the same type as ARG1.

**Result type and attributes**
The result is an INTEGER(4).

**Result value**
The result is 1 if each element of ARG1 is not greater than or equal to the corresponding element of ARG2. Otherwise, the result is 0.

---

**VEC_ALL_NGT(ARG1, ARG2)**

**Purpose**
Tests whether each element of the first argument is not greater than the corresponding element of the second argument.

**Class**
Elemental function

**Argument type and attributes**

**ARG1**
An INTENT(IN) real vector.
ARG2
An INTENT(IN) vector of the same type as ARG1.

Result type and attributes
The result is an INTEGER(4).

Result value
The result is 1 if each element of ARG1 is not greater than the corresponding element of ARG2. Otherwise, the result is 0.

VEC_ALL_NLE(ARG1, ARG2)

Purpose
Tests whether each element of the first argument is not less than or equal to the corresponding element of the second argument.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) real vector.

ARG2
An INTENT(IN) vector of the same type as ARG1.

Result type and attributes
The result is an INTEGER(4).

Result value
The result is 1 if each element of ARG1 is not less than or equal to the corresponding element of ARG2. Otherwise, the result is 0.

VEC_ALL_NLT(ARG1, ARG2)

Purpose
Tests whether each element of the first argument is not less than the corresponding element of the second argument.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) real vector.

ARG2
An INTENT(IN) vector of the same type as ARG1.
**Result type and attributes**

The result is an INTEGER(4).

**Result value**

The result is 1 if each element of ARG1 is not less than the corresponding element of ARG2. Otherwise, the result is 0.

---

**VEC_ALL_NUMERIC(ARG1)**

**Purpose**

Tests whether each element of the given vector is numeric (not a NaN).

**Class**

Elemental function

**Argument type and attributes**

ARG1

An INTENT(IN) real vector.

**Result type and attributes**

The result is an INTEGER(4).

**Result value**

The result is 1 if each element of ARG1 is numeric (not a NaN). Otherwise, the result is 0.

---

**VEC_AND(ARG1, ARG2)**

**Purpose**

Performs a bitwise AND of the given vectors.

**Class**

Elemental function

**Argument type and attributes**

ARG1

An INTENT(IN) integer vector, real vector, or unsigned vector.

ARG2

An INTENT(IN) vector of the same type as ARG1.

**Result type and attributes**

The result is of the same type as ARG1.

**Result value**

The result is the bitwise AND of ARG1 and ARG2.
VEC_ANDC(ARG1, ARG2)

**Purpose**
Performs a bitwise AND of the first argument and the bitwise complement of the second argument.

**Class**
Elemental function

**Argument type and attributes**

ARG1
An INTENT(IN) integer vector, real vector, or unsigned vector.

ARG2
An INTENT(IN) vector of the same type as ARG1.

**Result type and attributes**
The result is of the same type as ARG1.

**Result value**
The result is the bitwise AND of ARG1 with the bitwise complement of ARG2.

VEC_ANY_EQ(ARG1, ARG2)

**Purpose**
Tests whether any set of corresponding elements of the given vectors are equal.

**Class**
Elemental function

**Argument type and attributes**

ARG1
An INTENT(IN) vector.

ARG2
An INTENT(IN) vector of the same type as ARG1.

**Result type and attributes**
The result is an INTEGER(4).

**Result value**
The result is 1 if any element of ARG1 is equal to the corresponding element of ARG2. Otherwise, the result is 0.

VEC_ANY_GE(ARG1, ARG2)

**Purpose**
Tests whether any element of the first argument is greater than or equal to the corresponding element of the second argument.
Class
Elemental function

Argument type and attributes

ARG1
An INTENT(IN) integer vector, real vector, or unsigned vector.

ARG2
An INTENT(IN) vector of the same type as ARG1.

Result type and attributes
The result is an INTEGER(4).

Result value
The result is 1 if any element of ARG1 is greater than or equal to the corresponding element of ARG2. Otherwise, the result is 0.

VEC_ANY_GT(ARG1, ARG2)

Purpose
Tests whether any element of the first argument is greater than the corresponding element of the second argument.

Class
Elemental function

Argument type and attributes

ARG1
An INTENT(IN) integer vector, real vector, or unsigned vector.

ARG2
An INTENT(IN) vector of the same type as ARG1.

Result type and attributes
The result is an INTEGER(4).

Result value
The result is 1 if any element of ARG1 is greater than the corresponding element of ARG2. Otherwise, the result is 0.

VEC_ANY_LE(ARG1, ARG2)

Purpose
Tests whether any element of the first argument is less than or equal to the corresponding element of the second argument.

Class
Elemental function
Argument type and attributes

ARG1
An INTENT(IN) integer vector, real vector, or unsigned vector.

ARG2
An INTENT(IN) vector of the same type as ARG1.

Result type and attributes
The result is an INTEGER(4).

Result value
The result is 1 if any element of ARG1 is less than or equal to the corresponding element of ARG2. Otherwise, the result is 0.

VEC_ANY_LT(ARG1, ARG2)

Purpose
Tests whether any element of the first argument is less than the corresponding element of the second argument.

Class
Elemental function

Argument type and attributes

ARG1
An INTENT(IN) integer vector, real vector, or unsigned vector.

ARG2
An INTENT(IN) vector of the same type as ARG1.

Result type and attributes
The result is an INTEGER(4).

Result value
The result is 1 if any element of ARG1 is less than the corresponding element of ARG2. Otherwise, the result is 0.

VEC_ANY_NAN(ARG1)

Purpose
Tests whether any element of the given vector is a NaN.

Class
Elemental function

Argument type and attributes

ARG1
An INTENT(IN) real vector.
Result type and attributes
The result is an INTEGER(4).

Result value
The result is 1 if any element of ARG1 is a NaN. Otherwise, the result is 0.

VEC_ANY_NE(ARG1, ARG2)

Purpose
Tests whether any set of corresponding elements of the given vectors are not equal.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) vector.
ARG2
An INTENT(IN) vector of the same type as ARG1.

Result type and attributes
The result is an INTEGER(4).

Result value
The result is 1 if any element of ARG1 is not equal to the corresponding element of ARG2. Otherwise, the result is 0.

VEC_ANY_NGE(ARG1, ARG2)

Purpose
Tests whether any element of the first argument is not greater than or equal to the corresponding element of the second argument.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) real vector.
ARG2
An INTENT(IN) vector of the same type as ARG1.

Result type and attributes
The result is an INTEGER(4).
**Result value**

The result is 1 if any element of ARG1 is not greater than or equal to the corresponding element of ARG2. Otherwise, the result is 0.

---

**VEC_ANY_NGT(ARG1, ARG2)**

**Purpose**

Tests whether any element of the first argument is not greater than the corresponding element of the second argument.

**Class**

Elemental function

**Argument type and attributes**

ARG1

An INTENT(IN) real vector.

ARG2

An INTENT(IN) vector of the same type as ARG1.

**Result type and attributes**

The result is an INTEGER(4).

**Result value**

The result is 1 if any element of ARG1 is not greater than the corresponding element of ARG2. Otherwise, the result is 0.

---

**VEC_ANY_NLE(ARG1, ARG2)**

**Purpose**

Tests whether any element of the first argument is not less than or equal to the corresponding element of the second argument.

**Class**

Elemental function

**Argument type and attributes**

ARG1

An INTENT(IN) real vector.

ARG2

An INTENT(IN) vector of the same type as ARG1.

**Result type and attributes**

The result is an INTEGER(4).

**Result value**

The result is 1 if any element of ARG1 is not less than or equal to the corresponding element of ARG2. Otherwise, the result is 0.
**VEC_ANY_NLT(ARG1, ARG2)**

**Purpose**
Tests whether any element of the first argument is not less than the corresponding element of the second argument.

**Class**
Elemental function

**Argument type and attributes**

ARG1
An INTENT(IN) real vector.

ARG2
An INTENT(IN) vector of the same type as ARG1.

**Result type and attributes**
The result is an INTEGER(4).

**Result value**
The result is 1 if any element of ARG1 is not less than the corresponding element of ARG2. Otherwise, the result is 0.

---

**VEC_ANY_NUMERIC(ARG1)**

**Purpose**
Tests whether any element of the given vector is numeric (not a NaN).

**Class**
Elemental function

**Argument type and attributes**

ARG1
An INTENT(IN) real vector.

**Result type and attributes**
The result is an INTEGER(4).

**Result value**
The result is 1 if any element of ARG1 is numeric (not a NaN). Otherwise, the result is 0.

---

**VEC_ANY_OUT(ARG1, ARG2)**

**Purpose**
Tests whether the value of any element of a given vector is outside of a given range.
Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) real vector.
ARG2
An INTENT(IN) vector of the same type as ARG1.

Result type and attributes
The result is an INTEGER(4).

Result value
The result is 1 if the value of any element of ARG1 is greater than the value of the corresponding element of ARG2 or less than the negative of the value of the corresponding element of ARG2. Otherwise, the result is 0.

VEC_Avg(ARG1, ARG2)

Purpose
Returns a vector containing the average of each set of corresponding elements of the given vectors.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) integer vector or unsigned vector.
ARG2
An INTENT(IN) vector of the same type as ARG1.

Result type and attributes
The result is a vector of the same type as ARG1.

Result value
The value of each element of the result is the average of the values of the corresponding elements of ARG1 and ARG2.

VEC_CEIL(ARG1)

Purpose
Returns a vector containing the smallest representable floating-point integer values greater than or equal to the values of the corresponding elements of the given vector.
Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) real vector.

Result type and attributes
The result is a vector of the same type as ARG1.

Result value
Each element of the result contains the smallest representable floating-point integer
greater than or equal to the value of the corresponding element of ARG1.

VEC_CMPB(ARG1, ARG2)

Purpose
Performs a bounds comparison of each set of corresponding elements of the given
vectors.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) real vector.
ARG2
An INTENT(IN) vector of the same type as ARG1.

Result type and attributes
The result is an INTEGER(4) vector.

Result value
Each element of the result has the value 0 if the value of the corresponding
element of ARG1 is less than or equal to the value of the corresponding element of
ARG2 and greater than or equal to the negative of the value of the corresponding
element of ARG2. Otherwise:
• If an element of ARG2 is greater than or equal to zero, then the value of the
  corresponding element of the result is 0 if the absolute value of the
  corresponding element of ARG1 is equal to the value of the corresponding
  element of ARG2, negative if it is greater than the value of the corresponding
  element of ARG2, and positive if it is less than the value of the corresponding
  element of ARG2.
• If an element of ARG2 is less than zero, then the value of the element of the result
  is positive if the value of the corresponding element of ARG1 is less than or equal
  to the value of the element of ARG2, and negative otherwise.
VEC_CMPEQ(ARG1, ARG2)

Purpose
Returns a vector containing the results of comparing each set of corresponding elements of the given vectors for equality.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) integer vector, real vector, or unsigned vector.

ARG2
An INTENT(IN) vector of the same type as ARG1.

Result type and attributes
If ARG1 is an INTEGER(1) vector or UNSIGNED(1) vector, then the result is an UNSIGNED(1) vector. If ARG1 is an INTEGER(2) vector or UNSIGNED(2) vector, then the result is an UNSIGNED(2) vector. Otherwise, the result is an UNSIGNED(4) vector.

Result value
For each element of the result, the value of each bit is 1 if the corresponding elements of ARG1 and ARG2 are equal. Otherwise, the value of each bit is 0.

VEC_CMPGE(ARG1, ARG2)

Purpose
Returns a vector containing the results of a greater-than-or-equal-to comparison between each set of corresponding elements of the given vectors.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) real vector.

ARG2
An INTENT(IN) vector of the same type as ARG1.

Result type and attributes
The result is an UNSIGNED(4) vector.

Result value
For each element of the result, the value of each bit is 1 if the value of the corresponding element of ARG1 is greater than or equal to the value of the corresponding element of ARG2. Otherwise, the value of each bit is 0.
VEC_CMPGT(ARG1, ARG2)

Purpose
Returns a vector containing the results of a greater-than comparison between each set of corresponding elements of the given vectors.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) integer vector, real vector, or unsigned vector.
ARG2
An INTENT(IN) vector of the same type as ARG1.

Result type and attributes
If ARG1 is an INTEGER(1) vector or UNSIGNED(1) vector, then the result is an UNSIGNED(1) vector. If ARG1 is an INTEGER(2) vector or UNSIGNED(2) vector, then the result is an UNSIGNED(2) vector. Otherwise, the result is an UNSIGNED(4) vector.

Result value
For each element of the result, the value of each bit is 1 if the value of the corresponding element of ARG1 is greater than the value of the corresponding element of ARG2. Otherwise, the value of each bit is 0.

VEC_CMPLE(ARG1, ARG2)

Purpose
Returns a vector containing the results of a less-than-or-equal-to comparison between each set of corresponding elements of the given vectors.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) real vector.
ARG2
An INTENT(IN) vector of the same type as ARG1.

Result type and attributes
The result is an UNSIGNED(4) vector.

Result value
For each element of the result, the value of each bit is 1 if the value of the corresponding element of ARG1 is less than or equal to the value of the corresponding element of ARG2. Otherwise, the value of each bit is 0.
VEC_CMPLT(ARG1, ARG2)

Purpose
Returns a vector containing the results of a less-than comparison between each set of corresponding elements of the given vectors.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) integer vector, real vector, or unsigned vector.

ARG2
An INTENT(IN) vector of the same type as ARG1.

Result type and attributes
If ARG1 is an INTEGER(1) vector or UNSIGNED(1) vector, then the result is an UNSIGNED(1) vector. If ARG1 is an INTEGER(2) vector or UNSIGNED(2) vector, then the result is an UNSIGNED(2) vector. Otherwise, the result is an UNSIGNED(4) vector.

Result value
For each element of the result, the value of each bit is 1 if the value of the corresponding element of ARG1 is less than the value of the corresponding element of ARG2. Otherwise, the value of each bit is 0.

VEC_CONVERT(V, MOLD)

Purpose
Converts a vector to a vector of a given type.

Class
Pure function

Argument type and attributes
V  must be an INTENT(IN) vector.

MOLD
must be an INTENT(IN) vector. If it is a variable, it need not be defined.

Result type and attributes
The result is a vector of the same type as MOLD.

Result value
The result is as if it were on the left-hand side of an intrinsic assignment with V on the right-hand side.

Note: This intrinsic function can be used in initialization and specification expressions. It is also the only vector intrinsic function that is not in the C Altivec specification.
VEC_CTF(ARG1, ARG2)

**Purpose**
Converts a vector of fixed-point numbers into a vector of floating-point numbers.

**Class**
Elemental function

**Argument type and attributes**
ARG1
An INTENT(IN) INTEGER(4) vector or UNSIGNED(4) vector.

ARG2
An INTENT(IN) integer. Its value must be specified by an initialization expression whose value is between 0 and 31 inclusive.

**Result type and attributes**
The result is a REAL(4) vector.

**Result value**
The value of each element of the result is the closest floating-point estimate of the value of the corresponding element of ARG1 divided by 2 to the power of ARG2.

VEC_CTS(ARG1, ARG2)

**Purpose**
Converts a vector of floating-point numbers into a vector of signed fixed-point numbers.

**Class**
Elemental function

**Argument type and attributes**
ARG1
must be an INTENT(IN) real vector.

ARG2
must be an INTENT(IN) integer. Its value must be specified by an initialization expression whose value is between 0 and 31 inclusive.

**Result type and attributes**
The result is an INTEGER(4) vector.

**Result value**
The value of each element of the result is the saturated value obtained by multiplying the corresponding element of ARG1 by 2 to the power of ARG2.
VEC_CTU(ARG1, ARG2)

Purpose
Converts a vector of floating-point numbers into a vector of unsigned fixed-point numbers.

Class
Elemental function

Argument type and attributes
 ARG1
must be an INTENT(IN) real vector.

ARG2
must be an INTENT(IN) integer. Its value must be specified by an initialization expression whose value is between 0 and 31 inclusive.

Result type and attributes
The result is an UNSIGNED(4) vector.

Result value
The value of each element of the result is the saturated value obtained by multiplying the corresponding element of ARG1 by 2 to the power of ARG2.

VEC_DSS(ARG1)

Purpose
Stops the specified data stream read.

Class
Subroutine

Argument type and attributes
 ARG1
An INTENT(IN) integer and must be specified by an initialization expression whose value is within the range of 0 through 3 inclusive.

Stops the data stream read specified by ARG1.

VEC_DSSALL

Purpose
Stops all data stream reads.

Class
Subroutine
VEC_DST(ARG1, ARG2, ARG3)

**Purpose**
Initiates the data read of a line into cache in a state most efficient for reading.

**Class**
Subroutine

**Argument type and attributes**

**ARG1**
An INTENT(IN) variable or constant of a vector type or of type INTEGER(1), INTEGER(2), INTEGER(4), or REAL(4).

**ARG2**
An INTENT(IN) integer.

**ARG3**
An INTENT(IN) integer and must be specified by an initialization expression whose value is within the range of 0 through 3 inclusive.

The data stream specified by ARG3 is read beginning at the address of ARG1 using the control word ARG2. Use of this intrinsic indicates that the specified data stream is relatively persistent in nature.

VEC_DSTST(ARG1, ARG2, ARG3)

**Purpose**
Initiates the data read of a line into cache in a state most efficient for writing.

**Class**
Subroutine

**Argument type and attributes**

**ARG1**
An INTENT(IN) variable or constant of a vector type or of type INTEGER(1), INTEGER(2), INTEGER(4), or REAL(4).

**ARG2**
An INTENT(IN) integer.

**ARG3**
An INTENT(IN) integer and must be specified by an initialization expression whose value is within the range of 0 through 3 inclusive.

The data stream specified by ARG3 is read beginning at the address of ARG1 using the control word ARG2. Use of this intrinsic indicates that the specified data stream is relatively persistent in nature.

VEC_DSTSTT(ARG1, ARG2, ARG3)

**Purpose**
Initiates the data read of a line into cache in a state most efficient for writing.
Class
Subroutine

Argument type and attributes

ARG1
An INTENT(IN) variable or constant of a vector type or of type INTEGER(1), INTEGER(2), INTEGER(4), or REAL(4).

ARG2
An INTENT(IN) integer.

ARG3
An INTENT(IN) integer and must be specified by an initialization expression whose value is within the range of 0 through 3 inclusive.
The data stream specified by ARG3 is read beginning at the address of ARG1 using the control word ARG2. Use of this intrinsic indicates that the specified data stream is relatively transient in nature.

VEC_DSTT(ARG1, ARG2, ARG3)

Purpose
Initiates the data read of a line into cache in a state most efficient for reading.

Class
Subroutine

Argument type and attributes

ARG1
An INTENT(IN) variable or constant of a vector type or of type INTEGER(1), INTEGER(2), INTEGER(4), or REAL(4).

ARG2
An INTENT(IN) integer.

ARG3
An INTENT(IN) integer and must be specified by an initialization expression whose value is within the range of 0 through 3 inclusive.
The data stream specified by ARG3 is read beginning at the address of ARG1 using the control word ARG2. Use of this intrinsic indicates that the specified data stream is relatively transient in nature.

VEC_EXPTE(ARG1)

Purpose
Returns a vector containing estimates of 2 raised to the value of the corresponding elements of the given vector.

Class
Elemental function
**Argument type and attributes**

ARG1
An INTENT(IN) real vector.

**Result type and attributes**
The result is a vector of the same type as ARG1.

**Result value**
Each element of the result contains the estimated value of 2 raised to the value of the corresponding element of ARG1.

---

**VEC_FLOOR(ARG1)**

**Purpose**
Returns a vector containing the largest representable floating-point integer values less than or equal to the values of the corresponding elements of the given vector.

**Class**
Elemental function

**Argument type and attributes**

ARG1
An INTENT(IN) real vector.

**Result type and attributes**
The result is a vector of the same type as ARG1.

**Result value**
Each element of the result contains the largest representable floating-point integer less than or equal to the value of the corresponding element of ARG1.

---

**VEC_LD(ARG1, ARG2)**

**Purpose**
Loads a vector from a given memory address.

**Class**
Function

**Argument type and attributes**

ARG1
An INTENT(IN) integer.

ARG2
An INTENT(IN) integer of type INTEGER(1), INTEGER(2), INTEGER(4), or REAL(4), or any vector type.
Result type and attributes
If ARG2 is a vector, then the result is of the same type as ARG2. Otherwise, the result is a vector containing elements of the same type as ARG2.

Result value
ARG1 is added to the address of ARG2, and the sum is truncated to a multiple of 16 bytes. The result is the contents of the 16 bytes of memory starting at this address.

VEC_LDE(ARG1, ARG2)

Purpose
Loads an element from a given memory address into a vector.

Class
Function

Argument type and attributes
ARG1
An INTENT(IN) integer.

ARG2
An INTENT(IN) integer of type INTEGER(1), INTEGER(2), INTEGER(4), or REAL(4).

Result type and attributes
The result is a vector containing elements of the same type as ARG2.

Result value
The effective address is the sum of ARG1 and the address of ARG2, truncated to a multiple of the size in bytes of an element of the result vector. The contents of memory at the effective address are loaded into the result vector at the byte offset corresponding to the four least significant bits of the effective address. The remaining portions of the result vector are undefined.

VEC_LDL(ARG1, ARG2)

Purpose
Loads a vector from a given memory address, and marks the data as Least Recently Used.

Class
Function

Argument type and attributes
ARG1
An INTENT(IN) integer.

ARG2
An INTENT(IN) integer of type INTEGER(1), INTEGER(2), INTEGER(4), REAL(4) or any vector type.
**Result type and attributes**

If ARG2 is a vector, then the result is of the same type as ARG2. Otherwise, the result is a vector containing elements of the same type as ARG2.

**Result value**

ARG1 is added to the address of ARG2, and the sum is truncated to a multiple of 16 bytes. The result is the contents of the 16 bytes of memory starting at this address. This data is marked as Least Recently Used.

---

**VEC_LOGE(ARG1)**

**Purpose**

Returns a vector containing estimates of the base-2 logarithms of the corresponding elements of the given vector.

**Class**

Elemental function

**Argument type and attributes**

ARG1

An INTENT(IN) real vector.

**Result type and attributes**

The result is a vector of the same type as ARG1.

**Result value**

Each element of the result contains the estimated value of the base-2 logarithm of the corresponding element of ARG1.

---

**VEC_LVSL(ARG1, ARG2)**

**Purpose**

Returns a vector useful for aligning non-aligned data.

**Class**

Function

**Argument type and attributes**

ARG1

An INTENT(IN) integer.

ARG2

An INTENT(IN) integer of type INTEGER(1), INTEGER(2), INTEGER(4), or REAL(4).

**Result type and attributes**

The result is an UNSIGNED(1) vector.
Result value
The first element of the result vector is the sum of ARG1 and the address of ARG2, modulo 16. Each successive element contains the previous element’s value plus 1.

VEC_LVSR(ARG1, ARG2)

Purpose
Returns a vector useful for aligning non-aligned data.

Class
Function

Argument type and attributes
ARG1
An INTENT(IN) integer.

ARG2
An INTENT(IN) integer of type INTEGER(1), INTEGER(2), INTEGER(4), or REAL(4).

Result type and attributes
The result is an UNSIGNED(1) vector.

Result value
The effective address is the sum of ARG1 and the address of ARG2, modulo 16. The first element of the result vector contains the value 16 minus the effective address. Each successive element contains the previous element’s value plus 1.

VEC_MADD(ARG1, ARG2, ARG3)

Purpose
Returns a vector containing the results of performing a fused multiply/add for each corresponding set of elements of the given vectors.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) real vector.

ARG2
An INTENT(IN) vector of the same type as ARG1.

ARG3
An INTENT(IN) vector of the same type as ARG1.

Result type and attributes
The result is of the same type as ARG1.
**Result value**

The value of each element of the result is the product of the values of the corresponding elements of ARG1 and ARG2, added to the value of the corresponding element of ARG3.

---

**VEC_MADDS(ARG1, ARG2)**

**Purpose**

Returns a vector containing the results of performing a saturated multiply-high-and-add operation for each corresponding set of elements of the given vectors.

**Class**

Elemental function

**Argument type and attributes**

ARG1

An INTENT(IN) INTEGER(2) vector.

ARG2

An INTENT(IN) vector of the same type as ARG1.

ARG3

An INTENT(IN) vector of the same type as ARG1.

**Result type and attributes**

The result is of the same type as ARG1.

**Result value**

For each element of the result, the value is produced in the following way: The values of the corresponding elements of ARG1 and ARG2 are multiplied. The value of the 17 most significant bits of this product is then added, using 16-bit-saturated addition, to the value of the corresponding element of ARG3.

---

**VEC_MAX(ARG1, ARG2)**

**Purpose**

Returns a vector containing the maximum value from each set of corresponding elements of the given vectors.

**Class**

Elemental function

**Argument type and attributes**

ARG1

An INTENT(IN) integer vector, real vector, or unsigned vector.

ARG2

An INTENT(IN) vector of the same type as ARG1.
**Result type and attributes**
The result is a vector of the same type as ARG1.

**Result value**
The value of each element of the result is the maximum of the values of the corresponding elements of ARG1 and ARG2.

**VEC_MERGEH(ARG1, ARG2)**

**Purpose**
Merges the most significant halves of two vectors.

**Class**
Elemental function

**Argument type and attributes**

ARG1
An INTENT(IN) vector.

ARG2
An INTENT(IN) vector of the same type as ARG1.

**Result type and attributes**
The result is of the same type as ARG1.

**Result value**
Assume that the elements of each vector are numbered beginning with 0. The even-numbered elements of the result are taken, in order, from the elements in the most significant 8 bytes of ARG1. The odd-numbered elements of the result are taken, in order, from the elements in the most significant 8 bytes of ARG2.

**VEC_MERGEL(ARG1, ARG2)**

**Purpose**
Merges the least significant halves of two vectors.

**Class**
Elemental function

**Argument type and attributes**

ARG1
An INTENT(IN) vector.

ARG2
An INTENT(IN) vector of the same type as ARG1.

**Result type and attributes**
The result is of the same type as ARG1.
Result value

Assume that the elements of each vector are numbered beginning with 0. The even-numbered elements of the result are taken, in order, from the elements in the least significant 8 bytes of ARG1. The odd-numbered elements of the result are taken, in order, from the elements in the least significant 8 bytes of ARG2.

VEC_MFVSCR

Purpose
Copies the contents of the Vector Status and Control Register into the result vector.

Class
Function

Result type and attributes
The result is an UNSIGNED(2) vector.

Result value
The high-order 16 bits of the VSCR are copied into the seventh element of the result. The low-order 16 bits of the VSCR are copied into the eighth element of the result. All other elements are set to zero.

VEC_MIN(ARG1, ARG2)

Purpose
Returns a vector containing the minimum value from each set of corresponding elements of the given vectors.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) integer vector, real vector, or unsigned vector.

ARG2
An INTENT(IN) vector of the same type as ARG1.

Result type and attributes
The result is a vector of the same type as ARG1.

Result value
The value of each element of the result is the minimum of the values of the corresponding elements of ARG1 and ARG2.
VEC_MLADD(ARG1, ARG2, ARG3)

Purpose
Returns a vector containing the results of performing a saturated multiply-low-and-add operation for each corresponding set of elements of the given vectors.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) INTEGER(2) vector or UNSIGNED(2) vector.

ARG2
An INTENT(IN) INTEGER(2) vector or UNSIGNED(2) vector.

ARG3
An INTENT(IN) vector of the same type as ARG2.

Result type and attributes
If ARG1, ARG2, and ARG3 are all unsigned vectors, then the result is an UNSIGNED(2) vector. Otherwise, the result is an INTEGER(2) vector.

Result value
The value of each element of the result is the value of the least significant 16 bits of the product of the values of the corresponding elements of ARG1 and ARG2, added to the value of the corresponding element of ARG3.

The addition is performed using modular arithmetic.

VEC_MRADDS(ARG1, ARG2, ARG3)

Purpose
Returns a vector containing the results of performing a saturated multiply-high-round-and-add operation for each corresponding set of elements of the given vectors.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) INTEGER(2) vector.

ARG2
An INTENT(IN) vector of the same type as ARG1.

ARG3
An INTENT(IN) vector of the same type as ARG1.
Result type and attributes
The result is of the same type as ARG1.

Result value
For each element of the result, the value is produced in the following way: The values of the corresponding elements of ARG1 and ARG2 are multiplied and rounded such that the 15 least significant bits are 0. The value of the 17 most significant bits of this rounded product is then added, using 16-bit-saturated addition, to the value of the corresponding element of ARG3.

VEC_MSUM(ARG1, ARG2, ARG3)

Purpose
Returns a vector containing the results of performing a multiply-sum operation using the given vectors.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) INTEGER(1) vector, INTEGER(2) vector, UNSIGNED(1) vector, or UNSIGNED(2) vector.

ARG2
An INTENT(IN) vector. If ARG1 is an INTEGER(1) vector, then ARG2 is an UNSIGNED(1) vector. Otherwise, ARG2 is of the same type as ARG1.

ARG3
An INTENT(IN) vector. If ARG1 is an integer vector, then ARG3 is an INTEGER(4) vector. If ARG1 is an unsigned vector, then ARG3 is an UNSIGNED(4) vector.

Result type and attributes
The result is a vector of the same type as ARG3.

Result value
Assume that the elements of each vector are numbered beginning with 0. If ARG1 is an INTEGER(1) vector or an UNSIGNED(1) vector, then let m be 4. Otherwise, let m be 2. For each element n of the result vector, the value is obtained in the following way: For p = mn to mn+m-1, multiply element p of ARG1 by element p of ARG2. Add the sum of these products to element n of ARG3. All additions are performed using 32-bit modular arithmetic.

VEC_MSUMS(ARG1, ARG2, ARG3)

Purpose
Returns a vector containing the results of performing a saturated multiply-sum operation using the given vectors.

Class
Elemental function
**Argument type and attributes**

ARG1
An INTENT(IN) INTEGER(2) vector or UNSIGNED(2) vector.

ARG2
An INTENT(IN) vector of the same type as ARG1.

ARG3
An INTENT(IN) vector. If ARG1 is an integer vector, then ARG3 is an INTEGER(4) vector. If ARG1 is an unsigned vector, then ARG3 is an UNSIGNED(4) vector.

**Result type and attributes**
The result is a vector of the same type as ARG3.

**Result value**
Assume that the elements of each vector are numbered beginning with 0. For each element n of the result vector, the value is obtained in the following way: For p = 2n to 2n+1, multiply element p of ARG1 by element p of ARG2. Add the sum of these products to element n of ARG3. All additions are performed using 32-bit saturated arithmetic.

---

**VEC_MTVSCR(ARG1)**

**Purpose**
Copies the given value into the Vector Status and Control Register.

**Class**
Subroutine

**Argument type and attributes**

ARG1
An INTENT(IN) integer vector, unsigned vector, or pixel vector.

The low-order 32 bits of ARG1 are copied into the VSCR.

---

**VEC_MULE(ARG1, ARG2)**

**Purpose**
Returns a vector containing the results of multiplying every second corresponding set of elements of the given vectors, beginning with the first element.

**Class**
Elemental function

**Argument type and attributes**

ARG1
An INTENT(IN) INTEGER(1) vector, INTEGER(2) vector, UNSIGNED(1) vector, or UNSIGNED(2) vector.

ARG2
An INTENT(IN) vector of the same type as ARG1.
Result type and attributes
If ARG1 is an INTEGER(1) vector, then the result is an INTEGER(2) vector. If ARG1 is an INTEGER(2) vector, then the result is an INTEGER(4) vector. If ARG1 is an UNSIGNED(1) vector, then the result is an UNSIGNED(2) vector. If ARG1 is an UNSIGNED(2) vector, then the result is an UNSIGNED(4) vector.

Result value
Assume that the elements of each vector are numbered beginning with 0. For each element \( n \) of the result vector, the value is the product of the value of element \( 2n \) of ARG1 and the value of element \( 2n \) of ARG2.

VEC_MULO(ARG1, ARG2)

Purpose
Returns a vector containing the results of multiplying every second corresponding set of elements of the given vectors, beginning with the second element.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) INTEGER(1) vector, INTEGER(2) vector, UNSIGNED(1) vector, or UNSIGNED(2) vector.

ARG2
An INTENT(IN) vector of the same type as ARG1.

Result type and attributes
If ARG1 is an INTEGER(1) vector, then the result is an INTEGER(2) vector. If ARG1 is an INTEGER(2) vector, then the result is an INTEGER(4) vector. If ARG1 is an UNSIGNED(1) vector, then the result is an UNSIGNED(2) vector. If ARG1 is an UNSIGNED(2) vector, then the result is an UNSIGNED(4) vector.

Result value
Assume that the elements of each vector are numbered beginning with 0. For each element \( n \) of the result vector, the value is the product of the value of element \( 2n+1 \) of ARG1 and the value of element \( 2n+1 \) of ARG2.

VEC_NMSUB(ARG1, ARG2, ARG3)

Purpose
Returns a vector containing the results of performing a negative multiply-subtract operation on the given vectors.

Class
Elemental function
**Argument type and attributes**

ARG1
An INTENT(IN) real vector.

ARG2
An INTENT(IN) vector of the same type as ARG1.

ARG3
An INTENT(IN) vector of the same type as ARG1.

**Result type and attributes**
The result is a vector of the same type as ARG1.

**Result value**
The value of each element of the result is the product of the corresponding elements of ARG1 and ARG2, subtracted from the corresponding element of ARG3.

---

**VEC_NOR(ARG1, ARG2)**

**Purpose**
Performs a bitwise NOR of the given vectors.

**Class**
Elemental function

**Argument type and attributes**

ARG1
An INTENT(IN) integer vector, real vector, or unsigned vector.

ARG2
An INTENT(IN) vector of the same type as ARG1.

**Result type and attributes**
The result is of the same type as ARG1.

**Result value**
The result is the bitwise NOR of ARG1 and ARG2.

---

**VEC_OR(ARG1, ARG2)**

**Purpose**
Performs a bitwise OR of the given vectors.

**Class**
Elemental function

**Argument type and attributes**

ARG1
An INTENT(IN) integer vector, real vector, or unsigned vector.
ARG2
An INTENT(IN) vector of the same type as ARG1.

Result type and attributes
The result is of the same type as ARG1.

Result value
The result is the bitwise OR of ARG1 and ARG2.

---

VEC_PACK(ARG1, ARG2)

Purpose
Packs information from each element of two vectors into the result vector.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) integer vector or unsigned vector whose elements are of kind 2 or 4.
ARG2
An INTENT(IN) vector of the same type as ARG1.

Result type and attributes
If ARG1 is an INTEGER(2) vector, then the result is an INTEGER(1) vector. If ARG1 is an INTEGER(4) vector, then the result is an INTEGER(2) vector. If ARG1 is an UNSIGNED(2) vector, then the result is an UNSIGNED(1) vector. If ARG1 is an UNSIGNED(4) vector, then the result is an UNSIGNED(2) vector.

Result value
The value of each element of the result vector is taken from the low-order half of the corresponding element of the result of concatenating ARG1 and ARG2.

---

VEC_PACKPX(ARG1, ARG2)

Purpose
Packs information from each element of two vectors into the result vector.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) UNSIGNED(4) vector.
ARG2
An INTENT(IN) vector of the same type as ARG1.
Result type and attributes

The result is a pixel vector.

Result value

The value of each element of the result vector is taken from the corresponding element of the result of concatenating ARG1 and ARG2 in the following way: the least significant bit of the high order byte is stored into the first bit of the result element; the least significant 5 bits of each of the remaining bytes are stored into the remaining portion of the result element.

VEC_PACKS(ARG1, ARG2)

Purpose

Packs information from each element of two vectors into the result vector, using saturated values.

Class

Elemental function

Argument type and attributes

ARG1

An INTENT(IN) integer vector or unsigned vector whose elements are of kind 2 or 4.

ARG2

An INTENT(IN) vector of the same type as ARG1.

Result type and attributes

If ARG1 is an INTEGER(2) vector, then the result is an INTEGER(1) vector.

If ARG1 is an INTEGER(4) vector, then the result is an INTEGER(2) vector.

If ARG1 is an UNSIGNED(2) vector, then the result is an UNSIGNED(1) vector.

If ARG1 is an UNSIGNED(4) vector, then the result is an UNSIGNED(2) vector.

Result value

The value of each element of the result vector is the saturated value of the corresponding element of the result of concatenating ARG1 and ARG2.

VEC_PACKSU(ARG1, ARG2)

Purpose

Packs information from each element of two vectors into the result vector, using saturated values.

Class

Elemental function
**Argument type and attributes**

**ARG1**
An INTENT(IN) integer or unsigned vector whose elements are of kind 2 or 4.

**ARG2**
An INTENT(IN) vector of the same type as ARG1.

**Result type and attributes**
If ARG1 is an INTEGER(2) vector or an UNSIGNED(2) vector, then the result is an UNSIGNED(1) vector. If ARG1 is an INTEGER(4) vector or an UNSIGNED(4) vector, then the result is an UNSIGNED(2) vector.

**Result value**
The value of each element of the result vector is the saturated value of the corresponding element of the result of concatenating ARG1 and ARG2.

---

**VEC_PERM(ARG1, ARG2, ARG3)**

**Purpose**
Returns a vector that contains some elements of two vectors, in the order specified by a third vector.

**Class**
Elemental function

**Argument type and attributes**

**ARG1**
An INTENT(IN) vector.

**ARG2**
An INTENT(IN) vector of the same type as ARG1.

**ARG3**
An INTENT(IN), UNSIGNED(1) vector.

**Result type and attributes**
The result is of the same type as ARG1.

**Result value**
Each byte of the result is selected by using the least significant 5 bits of the corresponding byte of ARG3 as an index into the concatenated bytes of ARG1 and ARG2.

---

**VEC_RE(ARG1)**

**Purpose**
Returns a vector containing estimates of the reciprocals of the corresponding elements of the given vector.
Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) real vector.

Result type and attributes
The result is a vector of the same type as ARG1.

Result value
Each element of the result contains the estimated value of the reciprocal of the corresponding element of ARG1.

VEC_RL(ARG1, ARG2)

Purpose
Rotates each element of a vector left by a given number of bits.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) integer vector or unsigned vector.

ARG2
An INTENT(IN) unsigned vector containing elements of the same kind as the elements of ARG1.

Result type and attributes
The result is of the same type as ARG1.

Result value
Each element of the result is obtained by rotating the corresponding element of ARG1 left by the number of bits specified by the corresponding element of ARG2.

VEC_ROUND(ARG1)

Purpose
Returns a vector containing the rounded values of the corresponding elements of the given vector.

Class
Elemental function
**Argument type and attributes**

ARG1

An INTENT(IN) real vector.

**Result type and attributes**

The result is a vector of the same type as ARG1.

**Result value**

Each element of the result contains the value of the corresponding element of ARG1, rounded to the nearest representable floating-point integer, using IEEE round-to-nearest rounding.

**VEC_RSQRTE(ARG1)**

**Purpose**

Returns a vector containing estimates of the reciprocal square roots of the corresponding elements of the given vector.

**Class**

Elemental function

**Argument type and attributes**

ARG1

An INTENT(IN) real vector.

**Result type and attributes**

The result is a vector of the same type as ARG1.

**Result value**

Each element of the result contains the estimated value of the reciprocal square root of the corresponding element of ARG1.

**VEC_SEL(ARG1, ARG2, ARG3)**

**Purpose**

Selectively merges two vectors.

**Class**

Elemental function

**Argument type and attributes**

ARG1

INTENT(IN) integer vector, real vector, or unsigned vector.

ARG2

An INTENT(IN) vector of the same type as ARG1.
ARG3
An INTENT(IN) unsigned vector containing elements with the same kind as the elements of ARG1.

Result type and attributes
The result is of the same type as ARG1.

Result value
Each bit of the result vector has the value of the corresponding bit of ARG1 if the corresponding bit of ARG3 is 0, or the value of the corresponding bit of ARG2 otherwise.

VEC_SL(ARG1, ARG2)

Purpose
Performs a left shift for each element of a vector.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) integer vector or unsigned vector.

ARG2
An INTENT(IN) unsigned vector containing elements of the same kind as the elements of ARG1.

Result type and attributes
The result is of the same type as ARG1.

Result value
Each element of the result vector is the result of left shifting the corresponding element of ARG1 by the number of bits specified by the value of the corresponding element of ARG2, modulo the number of bits in the element. The bits that are shifted out are replaced by zeroes.

VEC_SLD(ARG1, ARG2, ARG3)

Purpose
Left shifts two concatenated vectors by a given number of bytes.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) vector.
**ARG2**
An INTENT(IN) vector of the same type as ARG1.

**ARG3**
An INTENT(IN) integer. It must be specified by an initialization expression whose value is within the range of 0 through 15 inclusive.

**Result type and attributes**
The result is of the same type as ARG1.

**Result value**
The result is the most significant 16 bytes obtained by concatenating ARG1 and ARG2, and shifting left by the number of bytes specified by ARG3.

---

**VEC_SLL(ARG1, ARG2)**

**Purpose**
Left shifts a vector by a given number of bits.

**Class**
Elemental function

**Argument type and attributes**

**ARG1**
An INTENT(IN) integer vector, unsigned vector, or pixel vector.

**ARG2**
An INTENT(IN) unsigned vector. Each byte of ARG2 must contain the same value.

**Result type and attributes**
The result is of the same type as ARG1.

**Result value**
The result is the contents of ARG1, shifted left by the number of bits specified by the 3 least significant bits of ARG2. The bits that are shifted out are replaced by zeroes.

---

**VEC_SLO(ARG1, ARG2)**

**Purpose**
Left shifts a vector by a given number of bytes (octets).

**Class**
Elemental function

**Argument type and attributes**

**ARG1**
An INTENT(IN) vector.
ARG2
An INTENT(IN) unsigned vector containing elements of kind 1 or 2.

Result type and attributes
The result is of the same type as ARG1.

Result value
The result is the contents of ARG1, shifted left by the number of bytes specified by
bits 121 through 124 of ARG2. The bits that are shifted out are replaced by zeroes.

VEC_SPLAT(ARG1, ARG2)

Purpose
Returns a vector that has all of its elements set to a given value.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) vector.

ARG2
An INTENT(IN) integer. Its value must be specified by an initialization
expression that is greater than or equal to 0, and less than the number of
elements in ARG1.

Result type and attributes
The result is of the same type as ARG1.

Result value
Assume that the elements of ARG1 are numbered beginning with 0. The value of the
element of ARG1 specified by ARG2 is given to each element of the result vector.

VEC_SPLAT_S8(ARG1)

Purpose
Returns a vector with all elements equal to the given value.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) integer. Its value must be specified by an initialization
expression whose value is between -16 and 15 inclusive.

Result type and attributes
The result is an INTEGER(1) vector.
**Result value**

Each element of the result has the value of ARG1.

**VEC_SPLAT_S16(ARG1)**

**Purpose**

Returns a vector with all elements equal to the given value.

**Class**

Elemental function

**Argument type and attributes**

ARG1

An INTENT(IN) integer. Its value must be specified by an initialization expression whose value is between -16 and 15 inclusive.

**Result type and attributes**

The result is an INTEGER(2) vector.

**Result value**

Each element of the result has the value of ARG1.

**VEC_SPLAT_S32(ARG1)**

**Purpose**

Returns a vector with all elements equal to the given value.

**Class**

Elemental function

**Argument type and attributes**

ARG1

An INTENT(IN) integer. Its value must be specified by an initialization expression whose value is between -16 and 15 inclusive.

**Result type and attributes**

The result is an INTEGER(4) vector.

**Result value**

Each element of the result has the value of ARG1.

**VEC_SPLAT_U8(ARG1)**

**Purpose**

Returns a vector with all elements equal to the given value.
Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) integer. Its value must be specified by an initialization
expression whose value is between -16 and 15 inclusive.

Result type and attributes
The result is an UNSIGNED(1) vector.

Result value
The bit pattern of ARG1 is interpreted as an unsigned value. Each element of the
result is given this value.

VEC_SPLAT_U16(ARG1)

Purpose
Returns a vector with all elements equal to the given value.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) integer. Its value must be specified by an initialization
expression whose value is between -16 and 15 inclusive.

Result type and attributes
The result is an UNSIGNED(2) vector.

Result value
The bit pattern of ARG1 is interpreted as an unsigned value. Each element of the
result is given this value.

VEC_SPLAT_U32(ARG1)

Purpose
Returns a vector with all elements equal to the given value.

Class
Elemental function

Argument type and attributes
ARG1
An INTENT(IN) integer. Its value must be specified by an initialization
expression whose value is between -16 and 15 inclusive.
Result type and attributes
The result is an \texttt{UNSIGNED(4)} vector.

Result value
The bit pattern of \texttt{ARG1} is interpreted as an unsigned value. Each element of the result is given this value.

\texttt{VEC_SR(ARG1, ARG2)}

Purpose
Performs a right shift for each element of a vector.

Class
Elemental function

Argument type and attributes
\texttt{ARG1}
An \texttt{INTENT(IN)} integer vector or unsigned vector.

\texttt{ARG2}
An \texttt{INTENT(IN)} unsigned vector containing elements of the same kind as the elements of \texttt{ARG1}.

Result type and attributes
The result is of the same type as \texttt{ARG1}.

Result value
Each element of the result vector is the result of right shifting the corresponding element of \texttt{ARG1} by the number of bits specified by the value of the corresponding element of \texttt{ARG2}, modulo the number of bits in the element. The bits that are shifted out are replaced by zeroes.

\texttt{VEC_SRA(ARG1, ARG2)}

Purpose
Performs an algebraic right shift for each element of a vector.

Class
Elemental function

Argument type and attributes
\texttt{ARG1}
An \texttt{INTENT(IN)} integer vector or unsigned vector.

\texttt{ARG2}
An \texttt{INTENT(IN)} unsigned vector containing elements of the same kind as the elements of \texttt{ARG1}.
**Result type and attributes**

The result is of the same type as ARG1.

**Result value**

Each element of the result vector is the result of algebraically right shifting the corresponding element of ARG1 by the number of bits specified by the value of the corresponding element of ARG2, modulo the number of bits in the element. The bits that are shifted out are replaced by copies of the most significant bit of the element of ARG1.

---

**VEC_SRL(ARG1, ARG2)**

**Purpose**

Right shifts a vector by a given number of bits.

**Class**

Elemental function

**Argument type and attributes**

ARG1

An INTENT(IN) integer vector, unsigned vector, or pixel vector.

ARG2

An INTENT(IN) unsigned vector. Each byte of ARG2 must contain the same value.

**Result type and attributes**

The result is of the same type as ARG1.

**Result value**

The result is the contents of ARG1, shifted right by the number of bits specified by the 3 least significant bits of ARG2. The bits that are shifted out are replaced by zeroes.

---

**VEC_SRO(ARG1, ARG2)**

**Purpose**

Right shifts a vector by a given number of bytes (octets).

**Class**

Elemental function

**Argument type and attributes**

ARG1

An INTENT(IN) vector.

ARG2

An INTENT(IN) unsigned vector whose elements are of kind 1 or 2.
**Result type and attributes**

The result is of the same type as ARG1.

**Result value**

The result is the contents of ARG1, shifted right by the number of bytes specified by bits 121 through 124 of ARG2. The bits that are shifted out are replaced by zeroes.

---

**VEC_ST(ARG1, ARG2, ARG3)**

**Purpose**

Stores a vector into memory at the given address.

**Class**

Subroutine

**Argument type and attributes**

**ARG1**  
An INTENT(IN) vector.

**ARG2**  
An INTENT(IN) integer.

**ARG3**  
is INTENT(OUT). It must be a vector or be of type integer or real as follows:

Case (i)  
If ARG3 is a vector, it must be of the same type as ARG1.

Case (ii)  
If ARG3 is not a vector, and ARG1 is an integer vector or an unsigned vector, then ARG3 must be of type integer with the same kind type parameter as the elements of ARG1.

Case (iii)  
If ARG3 is not a vector, and ARG1 is a real vector, then ARG3 must be of the same type and kind as the elements of ARG1.

Case (iv)  
If ARG3 is not a vector, and ARG1 is a pixel vector, then ARG3 must be of type INTEGER(2).

ARG2 is added to the address of ARG3, and the sum is truncated to a multiple of 16 bytes. The value of ARG1 is then stored into this memory address.

---

**VEC_STE(ARG1, ARG2, ARG3)**

**Purpose**

Stores a vector element into memory at the given address.

**Class**

Subroutine
**Argument type and attributes**

ARG1
An INTENT(IN) vector.

ARG2
An INTENT(IN) integer.

ARG3
is INTENT(OUT). If ARG1 is a pixel vector, then ARG3 must be of type INTEGER(2). If ARG1 is an unsigned vector, then ARG3 must be of type integer and must have the same kind as the elements of ARG1. Otherwise, ARG3 must have the same type and kind as the elements of ARG1.

The effective address is the sum of ARG2 and the address of ARG3, truncated to a multiple of the size in bytes of an element of the result vector. The value of the element of ARG1 at the byte offset that corresponds to the four least significant bits of the effective address is stored into memory at the effective address.

**VEC_STL(ARG1, ARG2, ARG3)**

**Purpose**
Stores a vector into memory at the given address, and marks the data as Least Recently Used.

**Class**
Subroutine

**Argument type and attributes**

ARG1
An INTENT(IN) vector.

ARG2
An INTENT(IN) integer.

ARG3
is INTENT(OUT). It must be a vector or be of type integer or real as follows:

Case (i)
If ARG3 is a vector, it must be of the same type as ARG1.

Case (ii)
If ARG3 is not a vector, and ARG1 An integer vector or an unsigned vector, then ARG3 must be of type integer with the same kind type parameter as the elements of ARG1.

Case (iii)
If ARG3 is not a vector, and ARG1 is a real vector, then ARG3 must be of the same type and kind as the elements of ARG1.

Case (iv)
If ARG3 is not a vector, and ARG1 is a pixel vector, then ARG3 must be of type INTEGER(2).

ARG2 is added to the address of ARG3, and the sum is truncated to a multiple of 16 bytes. The value of ARG1 is then stored into this memory address. The data is marked as Least Recently Used.
**VEC_SUB(ARG1, ARG2)**

**Purpose**
Returns a vector containing the differences of each set of corresponding elements of the given vectors.

**Class**
Elemental function

**Argument type and attributes**
- **ARG1**
  An INTENT(IN) integer vector, unsigned vector, or real vector.
- **ARG2**
  An INTENT(IN) vector of the same type as ARG1.

**Result type and attributes**
The result is of the same type as ARG1.

**Result value**
The value of each element of the result is the result of subtracting the value of the corresponding element of ARG2 from the value of the corresponding element of ARG1. The arithmetic is modular for integer vectors.

---

**VEC_SUBC(ARG1, ARG2)**

**Purpose**
Returns a vector containing the carry produced by subtracting each set of corresponding elements of the given vectors.

**Class**
Elemental function

**Argument type and attributes**
- **ARG1**
  An INTENT(IN) UNSIGNED(4) vector.
- **ARG2**
  An INTENT(IN) vector of the same type as ARG1.

**Result type and attributes**
The result is of the same type as ARG1.

**Result value**
The value of each element of the result is the value of the carry produced by subtracting the value of the corresponding element of ARG2 from the value of the corresponding element of ARG1. The value is 0 if a borrow occurred, or 1 if no borrow occurred.
VEC_SUBS(ARG1, ARG2)

**Purpose**
Returns a vector containing the saturated differences of each set of corresponding elements of the given vectors.

**Class**
Elemental function

**Argument type and attributes**

ARG1
An INTENT(IN) integer vector or unsigned vector.

ARG2
An INTENT(IN) vector of the same type as ARG1.

**Result type and attributes**
The result is of the same type as ARG1.

**Result value**
The value of each element of the result is the saturated result of subtracting the value of the corresponding element of ARG2 from the value of the corresponding element of ARG1.

VEC_SUM2S(ARG1, ARG2)

**Purpose**
Returns a vector containing the results of performing a sum across 1/2 vector operation on the given vectors.

**Class**
Elemental function

**Argument type and attributes**

ARG1
An INTENT(IN) INTEGER(4) vector.

ARG2
An INTENT(IN) vector of the same type as ARG1.

**Result type and attributes**
The result is of the same type as ARG1.

**Result value**
The first and third element of the result are 0. The second element of the result contains the saturated sum of the first and second elements of ARG1 and the second element of ARG2. The fourth element of the result contains the saturated sum of the third and fourth elements of ARG1 and the fourth element of ARG2.
**VEC_SUM4S(ARG1, ARG2)**

**Purpose**

Returns a vector containing the results of performing a sum across 1/4 vector operation on the given vectors.

**Class**

Elemental function

**Argument type and attributes**

ARG1  
An INTENT(IN) INTEGER(1) vector, INTEGER(2) vector, or UNSIGNED(1) vector.

ARG2  
An INTENT(IN) vector. If ARG1 is an integer vector, then ARG2 is an INTEGER(4) vector. If ARG1 is an unsigned vector, then ARG2 is an UNSIGNED(4) vector.

**Result type and attributes**

The result is a vector of the same type as ARG2.

**Result value**

Assume that the elements of each vector are numbered beginning with 0. If ARG1 is an INTEGER(1) vector or an UNSIGNED(1) vector, then let m be 4. Otherwise, let m be 2. For each element n of the result vector, the value is obtained by adding elements mn through mn+m-1 of ARG1 and element n of ARG2 using saturated addition.

**VEC_SUMS(ARG1, ARG2)**

**Purpose**

Returns a vector containing the results of performing a sum across vector operation on the given vectors.

**Class**

Elemental function

**Argument type and attributes**

ARG1  
An INTENT(IN) INTEGER(4) vector.

ARG2  
An INTENT(IN) vector of the same type as ARG1.

**Result type and attributes**

The result is a vector of the same type as ARG1.

**Result value**

The first three elements of the result are 0. The fourth element is the saturated sum of all the elements of ARG1 and the fourth element of ARG2.
**VEC_TRUNC(ARG1)**

**Purpose**
Returns a vector containing the truncated values of the corresponding elements of the given vector.

**Class**
Elemental function

**Argument type and attributes**

ARG1
An INTENT(IN) real vector.

**Result type and attributes**
The result is a vector of the same type as ARG1.

**Result value**
Each element of the result contains the value of the corresponding element of ARG1, truncated to an integral value.

---

**VEC_UNPACKH(ARG1)**

**Purpose**
Unpacks the most significant ("high") half of a vector into a vector with larger elements.

**Class**
Elemental function

**Argument type and attributes**

ARG1
An INTENT(IN) INTEGER(1) vector, INTEGER(2) vector, or pixel vector.

**Result type and attributes**
If ARG1 is an INTEGER(1) vector, then the result is an INTEGER(2) vector. If ARG1 is an INTEGER(2) vector, then the result is an INTEGER(4) vector. If ARG1 is a pixel vector, then the result is an UNSIGNED(4) vector.

**Result value**
If ARG1 is an integer vector, then the value of each element of the result is the value of the corresponding element of the most significant half of ARG1. If ARG1 is a pixel vector, then the value of each element of the result is taken from the corresponding element of the most significant half of ARG1 as follows: all bits in the first byte of the element of the result are set to the value of the first bit of the element of ARG1; the least significant 5 bits of the second byte of the element of the result are set to the value of the next 5 bits in the element of ARG1; the least significant 5 bits of the third byte of the element of the result are set to the value of the next 5 bits in the element of ARG1; the least significant 5 bits of the fourth byte of the element of the result are set to the value of the next 5 bits in the element of ARG1.
VEC_UNPACKL(ARG1)

**Purpose**
Unpacks the least significant ("low") half of a vector into a vector with larger elements.

**Class**
Elemental function

**Argument type and attributes**
ARG1
An INTENT(IN) INTEGER(1) vector, INTEGER(2) vector, or pixel vector.

**Result type and attributes**
If ARG1 is an INTEGER(1) vector, then the result is an INTEGER(2) vector. If ARG1 is an INTEGER(2) vector, then the result is an INTEGER(4) vector. If ARG1 is a pixel vector, then the result is an UNSIGNED(4) vector.

**Result value**
If ARG1 is an integer vector, then the value of each element of the result is the value of the corresponding element of the least significant half of ARG1. If ARG1 is a pixel vector, then the value of each element of the result is taken from the corresponding element of the least significant half of ARG1 as follows: all bits in the first byte of the element of the result are set to the value of the first bit of the element of ARG1; the least significant 5 bits of the second byte of the element of the result are set to the value of the next 5 bits in the element of ARG1; the least significant 5 bits of the third byte of the element of the result are set to the value of the next 5 bits in the element of ARG1; the least significant 5 bits of the fourth byte of the element of the result are set to the value of the next 5 bits in the element of ARG1.

VEC_XOR(ARG1, ARG2)

**Purpose**
Performs a bitwise XOR of the given vectors.

**Class**
Elemental function

**Argument type and attributes**
ARG1
An INTENT(IN) integer vector, real vector, or unsigned vector.

ARG2
An INTENT(IN) vector of the same type as ARG1.

**Result type and attributes**
The result is of the same type as ARG1.
Result value

The result is the bitwise XOR of ARG1 and ARG2.

End of IBM Extension
Chapter 16. Language interoperability features

XL Fortran provides a standardized mechanism for interoperating with C based on the Fortran 2003 Standard. An entity is said to be interoperable if equivalent declarations of it can be made in the two languages. XL Fortran enforces interoperability for types, variables, and procedures. Interoperability with the C programming language allows portable access to many libraries and the low-level facilities provided by C and allows the portable use of Fortran libraries by programs written in C. XL The details of this implementation are discussed in this section.

Interoperability of types

Intrinsic types

XL Fortran provides the ISO_C_BINDING intrinsic module that contains named constants holding kind type parameter values for intrinsic types. Their names are shown together with the corresponding C types in Table 32 on page 682. Only those intrinsic types listed in the table are interoperable; other intrinsic types are not.

Derived types

XL Fortran provides the ability to define derived types that correspond to C struct types. A Fortran derived type is interoperable with a C struct type if all of the following conditions are met:

- The Fortran derived-type definition is given the BIND(C) attribute explicitly.
- The Fortran derived type and C struct type have the same number of components.
- The components of the Fortran derived type have types and type parameters that are interoperable with the types of the corresponding components of the C struct type.
- The components of the Fortran derived type and of the C struct type are declared in the same relative positions in their relative type definitions.

For example, the C type myctype, declared below, is interoperable with the Fortran type myftype, declared below.

```fortran
typedef struct {
    int m, n;
    float r;
} myctype;
USE, INTRINSIC :: ISO_C_BINDING
TYPE, BIND(C) :: MYFTYPE
    INTEGER(C_INT) :: I, J
    REAL(C_FLOAT) :: S
END TYPE MYFTYPE
```

Note that the names of the corresponding components of the derived type and the C struct type need not be the same; the names are not significant in determining whether the Fortran derived type and C struct type are interoperable.
There is no Fortran type that is interoperable with a C struct type that contains a bit field or that contains a flexible array member. There is no Fortran type that is interoperable with a C union type.

Notes:
1. A derived type with the BIND attribute cannot be a SEQUENCE type.
2. A component of a derived type with the BIND attribute must have interoperable type and type parameters, and cannot have the POINTER or ALLOCATABLE attribute.

Interoperability of Variables

A Fortran module variable that has the BIND attribute may interoperate with a C variable with external linkage.

There need not be an associated C entity for a module variable with the BIND attribute.

A scalar Fortran variable is interoperable, if its type and type parameters are interoperable and it has neither the POINTER nor the ALLOCATABLE attributes. An interoperable scalar Fortran variable is interoperable with a scalar C variable, if its type and type parameters are interoperable with the type of the C variable.

A Fortran array variable is interoperable, if its type and type parameters are interoperable, it is of explicit shape or assumed size, it is not zero-sized, and it does not have the POINTER or ALLOCATABLE attributes.

A Fortran array is interoperable with a C array, if its size is nonzero and
• Its rank is equal to one and an element of the array is interoperable with an element of the C array
• Its rank is greater than one and the base types of the two arrays are equivalent and each of the dimensions correspond.

Because C uses row-major arrays and Fortran uses column-major arrays, a C array’s dimensions must be the reverse of a Fortran array’s dimensions.

Interoperability of common blocks

A C variable with external linkage can interoperate with a [common block] that has the [BIND] attribute.

If a common block has the BIND attribute, it must have the BIND attribute and the same binding label in each scoping unit in which it is declared. A C variable with external linkage interoperates with a common block with the BIND attribute if:
• The C variable is of a struct type and the variables that are members of the common block are interoperable with corresponding components of the struct type, or
• The common block contains a single variable, and the variable is interoperable with the C variable.

There need not be an associated C entity for a common block with the BIND attribute.
Interoperability of procedures

A Fortran procedure is interoperable if its interface is interoperable. A Fortran procedure interface is interoperable if it has the \texttt{BIND} attribute. A Fortran procedure interface is interoperable with a C function prototype if:

- The interface has the \texttt{BIND} attribute.
- The interface describes a function whose result variable is a scalar that is interoperable with the result of the prototype, or the interface describes a subroutine, and the prototype has a result type of \texttt{void}.
- The number of dummy arguments of the interface is equal to the number of formal parameters of the prototype.
- Any dummy argument with the \texttt{VALUE} attribute is interoperable with the corresponding formal parameter of the prototype.
- Any dummy argument without the \texttt{VALUE} attribute corresponds to a formal parameter of the prototype that is of a pointer type, and the dummy argument is interoperable with an entity of the referenced type of the formal parameter.
- The prototype does not have variable arguments.

In the following example, the Fortran procedure interface:

\begin{verbatim}
INTERFACE
  FUNCTION Func(i, j, k, l, m) BIND(C)
    USE, INTRINSIC :: ISO_C_BINDING
    INTEGER(C_SHORT) :: Func
    INTEGER(C_INT), VALUE :: i
    REAL(C_DOUBLE) :: J
    INTEGER(C_INT) :: k, l(10)
    TYPE(C_PTR), VALUE :: m
  END FUNCTION Func
END INTERFACE
\end{verbatim}

is interoperable with the C function prototype:

\begin{verbatim}
short func(int i, double *j, int *k, int l[10], void *m);
\end{verbatim}

A C data pointer may correspond to a Fortran dummy argument of type \texttt{C_PTR} or to a Fortran scalar that does not have the \texttt{VALUE} attribute. In the example, the C pointers \texttt{j} and \texttt{k} correspond to the Fortran scalars \texttt{J} and \texttt{K}, respectively. The C pointer \texttt{m} corresponds to the Fortran dummy argument \texttt{M} of type \texttt{C_PTR}.

The ISO_C_BINDING module

The \texttt{ISO_C_BINDING} module provides access to named constants that represent kind type parameters of data representations compatible with C types, the derived type \texttt{C_PTR} corresponding to any C data pointer type, the derived type \texttt{C_FUNPTR} corresponding to any C function pointer type, and four procedures.

Constants for use as kind type parameters

Table 32 on page 682 shows the interoperability between Fortran intrinsic types and C types. A Fortran intrinsic type with particular kind type parameter values is interoperable with a C type if the type and kind type parameter value are listed in the same row as that C type; if the type is character, interoperability also requires that the length type parameter be omitted or be specified by an initialization expression whose value is one. A combination of Fortran type and type parameters that is interoperable with a C type listed in the table is also interoperable with any unqualified C type that is compatible with the listed C type.
### Table 32. Interoperable Fortran and C types

<table>
<thead>
<tr>
<th>Fortran Type</th>
<th>Named Constant (kind type parameter)</th>
<th>Value</th>
<th>C Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>C_SIGNED_CHAR</td>
<td>1</td>
<td>signed char</td>
</tr>
<tr>
<td></td>
<td>C_SHORT</td>
<td>2</td>
<td>short</td>
</tr>
<tr>
<td></td>
<td>C_INT</td>
<td>4</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>C_LONG</td>
<td>4 (with -q32)</td>
<td>long</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 (with -q64)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C_LONG_LONG</td>
<td>8</td>
<td>long long</td>
</tr>
<tr>
<td></td>
<td>C_SIZE_T</td>
<td>4 (with -q32)</td>
<td>size_t</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 (with -q64)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C_INTPTR_T</td>
<td>4 (with -q32)</td>
<td>intptr_t</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 (with -q64)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C_INTMAX_T</td>
<td>8</td>
<td>intmax_t</td>
</tr>
<tr>
<td></td>
<td>C_INT8_T</td>
<td>1</td>
<td>int8_t</td>
</tr>
<tr>
<td></td>
<td>C_INT16_T</td>
<td>2</td>
<td>int16_t</td>
</tr>
<tr>
<td></td>
<td>C_INT32_T</td>
<td>4</td>
<td>int32_t</td>
</tr>
<tr>
<td></td>
<td>C_INT64_T</td>
<td>8</td>
<td>int64_t</td>
</tr>
<tr>
<td></td>
<td>C_INT_LEAST8_T</td>
<td>1</td>
<td>int_least8_t</td>
</tr>
<tr>
<td></td>
<td>C_INT_LEAST16_T</td>
<td>2</td>
<td>int_least16_t</td>
</tr>
<tr>
<td></td>
<td>C_INT_LEAST32_T</td>
<td>4</td>
<td>int_least32_t</td>
</tr>
<tr>
<td></td>
<td>C_INT_LEAST64_T</td>
<td>8</td>
<td>int_least64_t</td>
</tr>
<tr>
<td></td>
<td>C_INT_FAST8_T</td>
<td>1</td>
<td>int_fast8_t</td>
</tr>
<tr>
<td></td>
<td>C_INT_FAST16_T</td>
<td>4</td>
<td>int_fast16_t</td>
</tr>
<tr>
<td></td>
<td>C_INT_FAST32_T</td>
<td>4</td>
<td>int_fast32_t</td>
</tr>
<tr>
<td></td>
<td>C_INT_FAST64_T</td>
<td>8</td>
<td>int_fast64_t</td>
</tr>
<tr>
<td>REAL</td>
<td>C_FLOAT</td>
<td>4</td>
<td>float</td>
</tr>
<tr>
<td></td>
<td>C_DOUBLE</td>
<td>8</td>
<td>double</td>
</tr>
<tr>
<td></td>
<td>C_LONG_DOUBLE</td>
<td>16</td>
<td>long double</td>
</tr>
<tr>
<td></td>
<td>C_FLOAT_COMPLEX</td>
<td>4</td>
<td>float _Complex</td>
</tr>
<tr>
<td></td>
<td>C_DOUBLE_COMPLEX</td>
<td>8</td>
<td>double _Complex</td>
</tr>
<tr>
<td></td>
<td>C_LONG_DOUBLE_COMPLEX</td>
<td>16</td>
<td>long double _Complex</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>C_BOOL</td>
<td>1</td>
<td>_Bool</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>C_CHAR</td>
<td>1</td>
<td>char</td>
</tr>
</tbody>
</table>

For example, the type integer with a kind type parameter of C_SHORT is interoperable with the C type short or any C type derived (via typedef) from short.

**Notes:**

1. The named constants in the ISO_C_BINDING module are of type INTEGER(4).
2. In order for any Fortran COMPLEX entity to be interoperable with a corresponding C _Complex entity appearing in C code compiled with gcc, the Fortran code must be compiled with `-qfloat=complexgcc`.
3. Fortran REAL(C_LONG_DOUBLE) and COMPLEX(C_LONG_DOUBLE_COMPLEX) entities are only interoperable with the corresponding C types if the C code is compiled with an option that enables 128-bit long doubles.
4. Fortran integer entities with kind type parameter values of C_LONG_LONG, C_INT64_T, C_INT_LEAST64_T, C_INT_FAST64_T, C_INTMAX_T are only interoperable with the corresponding C types if the C compiler supports long long int types (q_longlong in the XL C/C++ compiler).
Character constants

The following character constants are provided for compatibility with some commonly used C characters that are represented using escape sequences:

<table>
<thead>
<tr>
<th>Fortran Named Constant</th>
<th>Definition</th>
<th>C Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_NULL_CHAR</td>
<td>null character</td>
<td>‘\0’</td>
</tr>
<tr>
<td>C_ALERT</td>
<td>alert</td>
<td>‘\a’</td>
</tr>
<tr>
<td>C_BACKSPACE</td>
<td>backspace</td>
<td>‘\b’</td>
</tr>
<tr>
<td>C_FORM_FEED</td>
<td>form feed</td>
<td>‘\f’</td>
</tr>
<tr>
<td>C_NEW_LINE</td>
<td>new line</td>
<td>‘\n’</td>
</tr>
<tr>
<td>C_CARRIAGE_RETURN</td>
<td>carriage return</td>
<td>‘\r’</td>
</tr>
<tr>
<td>C_HORIZONTAL_TAB</td>
<td>horizontal tab</td>
<td>‘\t’</td>
</tr>
<tr>
<td>C_VERTICAL_TAB</td>
<td>vertical tab</td>
<td>‘\v’</td>
</tr>
</tbody>
</table>

Other constants

The constant C_NULL_PTR is of type C_PTR; it has the value of a C null data pointer. The constant C_NULL_FUNPTR is of type C_FUNPTR; it has the value of a C null function pointer.

Types

The type C_PTR is interoperable with any C data pointer type. The type C_FUNPTR is interoperable with any C function pointer type. They are both derived types with private components.

Procedures

A C procedure argument is often defined in terms of a C address. The ISO_C_BINDING module provides the following procedures. The C_ASSOCIATED function is provided so that Fortran programs can compare C addresses. The C_F_POINTER subroutine provides a means of associating a Fortran pointer with the target of a C pointer. The C_FUNLOC and C_LOC functions are provided so that Fortran applications can determine the appropriate value to use with C facilities.

C_ASSOCIATED(C_PTR_1[, C_PTR_2])

Purpose: Indicates the association status of C_PTR_1, or whether C_PTR_1 and C_PTR_2 are associated with the same entity.

Class: Inquiry function

Argument type and attributes:

C_PTR_1

Required; scalar of type C_PTR or C_FUNPTR.

C_PTR_2

Optional; scalar of the same type as C_PTR_1.

Result type and attributes: Default logical

Result value:

• If C_PTR_2 is absent, then the result is false if C_PTR_1 is a C null pointer; otherwise, it has a value of true.
• If C_PTR_2 is present, then the result is false if C_PTR_1 is a C null pointer. Otherwise, the result is true if C_PTR_1 compares equal to C_PTR_2, and false otherwise.

C_F_POINTER(CPTR, FPTR [, SHAPE])

Purpose: Associates a data pointer with the target of a C pointer and specifies its shape.

Class: Subroutine

Argument type and attributes:

CPTR  An INTENT(IN) argument; a scalar and of type C_PTR.

FPTR  An INTENT(OUT) argument that is a pointer.

SHAPE An optional INTENT(IN) argument of type integer and rank one. If present, its size equals the rank of FPTR. SHAPE must be present if and only if FPTR is an array.

Rules: If the value of CPTR is the C address of an interoperable data entity, then:
• FPTR has type and type parameters that are interoperable with the type of the entity.
• FPTR becomes pointer associated with the target of CPTR.
• If FPTR is an array, its shape is specified by SHAPE, and each lower bound is 1.

Otherwise, the value of CPTR will be the result of a reference to C_LOC with a noninteroperable argument X. X (or its target) cannot have been deallocated or have become undefined due to the execution of a RETURN or END statement since the reference to C_LOC. FPTR is a nonpolymorphic, scalar pointer with the same type and type parameters as X. It becomes pointer-associated with X (or its target if X is a pointer).

C_FUNLOC(X)

Purpose: Returns the C address of a function pointer.

Class: Inquiry function

Argument type and attributes:

X  required interoperable procedure

Result type and attributes: Scalar of type C_FUNPTR

Result value: A value of type C_FUNPTR that represents the C address of the argument.

C_LOC(X)

Purpose: Returns the C address of the argument.

Class: Inquiry function

Argument type and attributes:

X  required and can be one of the following:
• an interoperable, nonpointer, nonallocatable data variable with the
  TARGET attribute.
• an allocated allocatable data variable with the TARGET attribute and
  interoperable type and type parameters and not a zero-sized array.
• an associated scalar pointer with interoperable type and type
  parameters.
• a nonallocatable, nonpointer, scalar variable that has the TARGET
  attribute and no nonkind type parameters.
• an allocated, nonpolymorphic, allocatable scalar pointer that has the
  TARGET attribute and no nonkind type parameters.
• an associated, nonpolymorphic, scalar pointer that has no nonkind type
  parameters.

Result type and attributes: Scalar of type C_PTR

Result value: A value of type C_PTR that represents the C address of the
argument.

Binding labels

A binding label is a value of type default character that specifies the name by
which a variable, common block, or a procedure is known to the C compiler.

If a variable, common block, or non-dummy procedure has the BIND attribute
specified with a NAME= specifier, the binding label is the value of the expression
specified for the NAME= specifier. The case of letters in the binding label is
significant, but leading and trailing blanks are ignored. If the entity has the BIND
attribute specified without a NAME= specifier, the binding label is the same as the
name of the entity using lower case letters.

The binding label of a C entity with external linkage is the same as the name of
the C entity. A Fortran entity with the BIND attribute that has the same binding
label as a C entity with external linkage is associated with that entity.

A binding label cannot be the same as another binding label or a name used to
identify any global entity of the Fortran program, ignoring differences in case
when-qmixed (or -U) is specified.

End of Fortran 2003 Standard
Chapter 17. The ISO_FORTRAN_ENV intrinsic module

The ISO_FORTRAN_ENV intrinsic module provides constants relating to the Fortran environment as follows:

- CHARACTER_STORAGE_SIZE
- ERROR_UNIT
- FILE_STORAGE_SIZE
- INPUT_UNIT
- IOSTAT_END
- IOSTAT_EOR
- NUMERIC_STORAGE_SIZE
- OUTPUT_UNIT

Note: The kind of the constants in this module, and the value of the NUMERIC_STORAGE_SIZE constant assume a default integer size of 4.

**CHARACTER_STORAGE_SIZE**

**Purpose**

The size, expressed in bits, of the character storage unit.

**Type**

Default integer scalar.

**Value**

8

**ERROR_UNIT**

**Purpose**

Identifies the preconnected external unit used for error reporting.

**Type**

Default integer scalar.

**Value**

0

**FILE_STORAGE_SIZE**

**Purpose**

The size, expressed in bits, of the file storage unit.
<table>
<thead>
<tr>
<th>Type</th>
<th>Default integer scalar.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>8</td>
</tr>
</tbody>
</table>

**INPUT_UNIT**

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Identifies the preconnected external unit used for input.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Default integer scalar.</td>
</tr>
<tr>
<td>Value</td>
<td>5</td>
</tr>
</tbody>
</table>

**IOSTAT_END**

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Assigned to the variable specified in an IOSTAT= specifier if an end-of-file condition occurs during execution of a [READ] statement. You must set the IOSTAT_END=2003std runtime option to get this value for end-of-file conditions on internal files. (See the [IOSTAT_END] runtime option in the [XL Fortran Compiler Reference] for more information.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Default integer scalar.</td>
</tr>
<tr>
<td>Value</td>
<td>-1</td>
</tr>
</tbody>
</table>

**IOSTAT_EOR**

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Assigned to the variable specified in an IOSTAT= specifier if an end-of-record condition occurs during execution of a [READ] statement.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Default integer scalar.</td>
</tr>
<tr>
<td>Value</td>
<td>-4</td>
</tr>
</tbody>
</table>
NUMERIC_STORAGE_SIZE

Purpose
The size, expressed in bits, of the numeric storage unit.

Type
Default integer scalar.

Value
32

OUTPUT_UNIT

Purpose
Identifies the preconnected external unit used for output.

Type
Default integer scalar.

Value
6

End of Fortran 2003 Standard
Chapter 18. Floating-point control and inquiry procedures

XL Fortran provides several ways that allow you to query and control the floating-point status and control register of the processor directly. These include:

- `fpgets` and `fpsets` subroutines
- Efficient floating-point control and inquiry procedures
- IEEE floating-point procedures, as specified in the Fortran 2003 Standard

The `fpgets` and `fpsets` subroutines retrieve and set the status of floating-point operations, respectively. Instead of calling operating system routines directly, these subroutines use an array of logicals named `fpstat` to pass information back and forth.

XL Fortran also provides procedures in the `xlf_fp_util` module that allow you to control the floating-point status and control register of the processor directly. These procedures are more efficient than the `fpgets` and `fpsets` subroutines; they are mapped into inlined machine instructions that directly manipulate the floating-point status and control register.

XL Fortran includes the `IEEE_ARITHMETIC`, `IEEE_EXCEPTIONS`, and `IEEE_FEATURES` modules to take advantage of the Fortran 2003 Standard rules for the IEEE floating-point status semantics.

### fpgets fpsets

The `fpgets` and `fpsets` subroutines retrieve and set the status of the floating-point operations, respectively. The include file `/usr/include/fpdc.h` contains the data declarations (specification statements) for the two subroutines. The include file `/usr/include/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 `fpgets` and `fpsets` subroutines in the `XL Fortran Optimization and Programming Guide` for examples and information on the elements of the `fpstat` array.

**Note:** The `XLF_FP_UTIL` intrinsic 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 692.

**Examples**

```fortran
CALL fpgets( fpstat )

...  

CALL fpsets( fpstat )
```
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 a procedure. The following files are supplied for the xlf_fp_util module:

<table>
<thead>
<tr>
<th>File names</th>
<th>File type</th>
<th>Locations</th>
</tr>
</thead>
<tbody>
<tr>
<td>xlf_fp_util.mod</td>
<td>module symbol file</td>
<td>install path/xlf/10.1/include</td>
</tr>
</tbody>
</table>

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 414.

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

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" on page 693
- "get_fpscr" on page 694
- "get_fpscr_flags" on page 694
- "get_round_mode" on page 695
- "set_fpscr" on page 695
- "set_fpscr_flags" on page 695
- "set_round_mode" on page 696

The following table lists the constants that are used with the fpscr procedures:

<table>
<thead>
<tr>
<th>Family</th>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>FPSCR_KIND</td>
<td>The kind type parameter for a fpscr flags variable</td>
</tr>
<tr>
<td>IEEE Rounding Modes</td>
<td>FP_RND_RN</td>
<td>Round toward nearest (default)</td>
</tr>
<tr>
<td></td>
<td>FP_RND_RZ</td>
<td>Round toward zero</td>
</tr>
<tr>
<td></td>
<td>FP_RND_RP</td>
<td>Round toward plus infinity</td>
</tr>
<tr>
<td></td>
<td>FP_RND_RM</td>
<td>Round toward minus infinity</td>
</tr>
<tr>
<td></td>
<td>FP_RND_MODE</td>
<td>Used to obtain the rounding mode from an FPSCR flags variable or value</td>
</tr>
<tr>
<td>Family</td>
<td>Constant</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>IEEE Exception Enable Flags</td>
<td>TRP_INEXACT</td>
<td>Enable inexact trap</td>
</tr>
<tr>
<td></td>
<td>TRP_DIV_BY_ZERO</td>
<td>Enable divide-by-zero trap</td>
</tr>
<tr>
<td></td>
<td>TRP_UNDERFLOW</td>
<td>Enable underflow trap</td>
</tr>
<tr>
<td></td>
<td>TRP_OVERFLOW</td>
<td>Enable overflow trap</td>
</tr>
<tr>
<td></td>
<td>TRP_INVALID</td>
<td>Enable invalid trap</td>
</tr>
<tr>
<td></td>
<td>FP_ENBL_SUMM</td>
<td>Trap enable summary or enable all</td>
</tr>
<tr>
<td>IEEE Exception Status Flags</td>
<td>FP_INVALID</td>
<td>Invalid operation exception</td>
</tr>
<tr>
<td></td>
<td>FP_OVERFLOW</td>
<td>Overflow exception</td>
</tr>
<tr>
<td></td>
<td>FP_UNDERFLOW</td>
<td>Underflow exception</td>
</tr>
<tr>
<td></td>
<td>FP_DIV_BY_ZERO</td>
<td>Divide-by-zero exception</td>
</tr>
<tr>
<td></td>
<td>FP_INEXACT</td>
<td>Inexact exception</td>
</tr>
<tr>
<td></td>
<td>FP_ALL_IEEE_XCP</td>
<td>All IEEE exceptions summary flags</td>
</tr>
<tr>
<td></td>
<td>FP_COMMON_IEEE_XCP</td>
<td>All IEEE exceptions summary flags excluding the FP_INEXACT exception</td>
</tr>
<tr>
<td>Machine Specific Exception Details Flags</td>
<td>FP_INV_SNAN</td>
<td>Signaling NaN</td>
</tr>
<tr>
<td></td>
<td>FP_INV_ISI</td>
<td>Infinity – Infinity</td>
</tr>
<tr>
<td></td>
<td>FP_INV_IDI</td>
<td>Infinity / Infinity</td>
</tr>
<tr>
<td></td>
<td>FP_INV_ZDZ</td>
<td>0 / 0</td>
</tr>
<tr>
<td></td>
<td>FP_INV_IMZ</td>
<td>Infinity * 0</td>
</tr>
<tr>
<td></td>
<td>FP_INV_CMP</td>
<td>Unordered compare</td>
</tr>
<tr>
<td></td>
<td>FP_INV_SQRT</td>
<td>Square root of negative number</td>
</tr>
<tr>
<td></td>
<td>FP_INV_CVI</td>
<td>Conversion to integer error</td>
</tr>
<tr>
<td></td>
<td>FP_INV_VXSOFT</td>
<td>Software request</td>
</tr>
<tr>
<td>Machine Specific Exception Summary Flags</td>
<td>FP_ANY_XCP</td>
<td>Any exception summary flag</td>
</tr>
<tr>
<td></td>
<td>FP_ALL_XCP</td>
<td>All exceptions summary flags</td>
</tr>
<tr>
<td></td>
<td>FP_COMMON_XCP</td>
<td>All exceptions summary flags excluding the FP_INEXACT exception</td>
</tr>
</tbody>
</table>

Notes:
1. In order to enable exception trapping, you must set the desired IEEE Exception Enable Flags and,
   • compile your program with the appropriate `-qflttrap` suboption. For more information on the `-qflttrap` compiler option and its suboptions, see the XLF Fortran Compiler Reference.

xlf_fp_util Floating-Point Procedures
This section lists the efficient floating-point control and inquiry procedures in the XLF_FP_UTIL intrinsic module.

clr_fpscr_flags
Type: 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 475.

For more information on the FPSCR constants, see "FPSCR constants" on page 692.

Examples:

```fortran
USE, INTRINSIC :: XLF_FP_UTIL
INTEGER(FPSCR_KIND) MASK

! Clear the overflow and underflow exception flags
MASK=(IOR(FP_OVERFLOW,FP_UNDERFLOW))
CALL clr_fpscr_flags(MASK)
```

For another example of the clr_fpscr_flags subroutine, see "get_fpscr_flags."

get_fpscr

**Type:** The get_fpscr function returns the current value of the floating-point status and control register (fpSCR) of the processor.

**Result type and attributes:** INTEGER(FPSCR_KIND)

**Result value:** The current value of the floating-point status and control register (FPSCR) of the processor.

Examples:

```fortran
USE, INTRINSIC :: XLF_FP_UTIL
INTEGER(FPSCR_KIND) FPSCR

FPSCR=get_fpscr()
```

get_fpscr_flags

**Type:** 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 475.

For more information on the FPSCR constants, see "FPSCR constants" on page 692.

**Result type and attributes:** An INTEGER(FPSCR_KIND)

**Result value:** The status of the FPSCR flags specified by the MASK argument. If a flag specified in the MASK argument is on, the value for the flag will be returned in the return value. The following example requests the status of the FP_DIV_BY_ZERO and FP_INVALID flags.

- If both flags are on, the return value is IOR(FP_DIV_BY_ZERO, FP_INVALID).
- If only the FP_INVALID flag is on, the return value is FP_INVALID.
- If only the FP_DIV_BY_ZERO flag is on, the return value is FP_DIV_BY_ZERO.
- If neither flag is on, the return value is 0.

Examples:

```fortran
USE, INTRINSIC :: 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.

... About processing the exception, the exception flags are cleared.
CALL clr_fpscr_flags(IOR(FP_DIV_BY_ZERO, FP_INVALID))
END IF

get_round_mode

Type: 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 692.

Result type and attributes: An INTEGER(FPSCR_KIND)

Result value: One of the constants FP_RND_RN, FP_RND_RZ, FP_RND_RP or FP_RND_RM.

Examples:
USE, INTRINSIC :: XLF_FP_UTIL
INTEGER(FPSCR_KIND) MODE'

MODE=get_round_mode()
IF (MODE .EQ. FP_RND_RZ) THEN
! ... 
END IF

set_fpscr

Type: 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.

Argument type and attributes: An INTEGER(FPSCR_KIND)

Result type and attributes: An INTEGER(FPSCR_KIND).

Result value: The value of the register before it was set with set_fpscr.

Examples:
USE, INTRINSIC :: XLF_FP_UTIL
INTEGER(FPSCR_KIND) FPSCR, OLD_FPSRC

FPSCR=get_fpscr()
! ... Some changes are made to FPSCR ...
OLD_FPSRC=set_fpscr(FPSCR) ! OLD_FPSRC is assigned the value of
! the register before it was
! set with set_fpscr

set_fpscr_flags

Type: 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 475.
For more information on the FPSCR constants, see “FPSCR constants” on page 692.

Examples:

**set_round_mode**

*Type:* 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 692.

*Argument type and attributes:* Integer of kind FPSCR_KIND

*Result type and attributes:* Integer of kind FPSCR_KIND

*Result value:* The rounding mode before the change.

**Examples:**

```fortran
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.
```

---

**IEEE Modules and support**

<table>
<thead>
<tr>
<th>Fortran 2003 Standard</th>
</tr>
</thead>
</table>

XL Fortran offers support for IEEE floating-point functionality as specified in the Fortran 2003 standard. The standard defines the IEEE_EXCEPTIONS module for exceptions, the IEEE_ARITHMETIC module to support IEEE arithmetic, and IEEE_FEATURES to specify the IEEE features supported by the compiler.

When using the IEEE_EXCEPTIONS, or IEEE_ARITHMETIC intrinsic modules, the XL Fortran compiler enforces several Fortran 2003 standard rules regarding the scope of changes to the floating-point status concerning rounding mode, halting mode, and exception flags. This can impede the performance of programs that use these modules, but do not utilize the new floating-point status semantics. For such programs, the `-qstrictieee` compiler option is provided to relax the rules on saving and restoring floating point status.

**Notes:**

1. XL Fortran Extended Precision floating-point numbers are not in the format suggested by the IEEE standard. As a result, some parts of the modules do not support REAL(16).
2. IEEE modules generate SIGFPE signals on Linux.

**Compiling and exception handling**

XL Fortran provides a number of options for strict compliance with the IEEE standard.

- Use `-qfloat=nomaf` to ensure compatibility with the IEEE standard for floating point arithmetic (IEEE 754-1985).
- When compiling programs that change the rounding mode, use `-qfloat=rrm`. 
• Use `-qfloat=nans` to detect signaling NaN values. Signaling NaN values can only occur if specified in a program.

• Use the `-qstrict` compiler option for strict conformance to the IEEE standard for floating-point arithmetic on programs compiled with an optimization level of -O3 or higher, `-qhot`, `-qipa`, `-qpdf`, or `-qsmp`.

**Related information**
For more information on IEEE floating-point and specific explanations of the compiler options listed above, see Implementation details of XL Fortran floating-point processing in the XL Fortran Optimization and Programming Guide.

**General rules for implementing IEEE modules**
The IEEE_ARITHMETIC, IEEE_EXCEPTIONS, and IEEE_FEATURES modules are intrinsic, though the types and procedures defined in these modules are not intrinsic.

All functions contained in IEEE modules are pure.

All procedure names are generic and not specific.

The default value for all exception flags is quiet.

By default, exceptions do not cause halting.

Rounding mode defaults towards nearest.

**IEEE Derived data types and constants**
The IEEE modules define the following derived types.

**IEEE_FLAG_TYPE**

*Type:* A derived data type defined by the IEEE_EXCEPTIONS module that identifies a particular exception flag. The values for IEEE_FLAG_TYPE must be one of the following named constants as defined in the IEEE_EXCEPTIONS module:

**IEEE_OVERFLOW**

Occurs when the result for an intrinsic real operation or an assignment has an exponent too large to be represented. This exception also occurs when the real or imaginary part of the result for an intrinsic complex operation or assignment has an exponent too large to be represented.

When using REAL(4), an overflow occurs when the result value’s unbiased exponent is > 127 or < -126.

When using REAL(8), an overflow occurs when the result value’s unbiased exponent is > 1023 or < -1022.

**IEEE_DIVIDE_BY_ZERO**

Occurs when a real or complex division has a nonzero numerator and a zero denominator.

**IEEE_INVALID**

Occurs when a real or complex operation or assignment is invalid.

**IEEE_UNDERFLOW**

Occurs when the result for an intrinsic real operation or assignment has an absolute value too small to be represented by anything other than zero,
and loss of accuracy is detected. The exception also occurs when the real or imaginary part of the result for an intrinsic complex operation or assignment has an absolute value that is too small to be represented by anything other than zero, and loss of accuracy is detected.

For REAL(4), an underflow occurs when the result has an absolute value $< 2^{149}$.

For REAL(8), an underflow occurs when the result has an absolute value $< 2^{1074}$.

**IEEE_INEXACT**

Occurs when the result of a real or complex assignment or operation is not exact.

The following constants are arrays of **IEEE_FLAG_TYPE**:

**IEEE_USUAL**

An array named constant containing **IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, and IEEE_INVALID** elements in order.

**IEEE_ALL**

An array named constant containing **IEEE_USUAL, IEEE_UNDERFLOW, and IEEE_INEXACT** elements in order.

**IEEE_STATUS_TYPE**

Type: A derived data type defined in the **IEEE_ARITHMETIC** module that represents the current floating-point status. The floating-point status encompasses the values of all exception flags, halting, and rounding modes.

**IEEE_CLASS_TYPE**

Type: A derived data type defined in the **IEEE_ARITHMETIC** module that categorizes a class of floating-point values. The values for **IEEE_CLASS_TYPE** must be one of the following named constants as defined in the **IEEE_ARITHMETIC** module:

<table>
<thead>
<tr>
<th>IEEE_SIGNALING_NAN</th>
<th>IEEE_NEGATIVE_ZERO</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE_QUIET_NAN</td>
<td>IEEE_POSITIVE_ZERO</td>
</tr>
<tr>
<td>IEEE_NEGATIVE_INF</td>
<td>IEEE_POSITIVE_DENORMAL</td>
</tr>
<tr>
<td>IEEE_NEGATIVE_NORMAL</td>
<td>IEEE_POSITIVE_NORMAL</td>
</tr>
<tr>
<td>IEEE_NEGATIVE_DENORMAL</td>
<td>IEEE_POSITIVE_INF</td>
</tr>
</tbody>
</table>

**IEEE_ROUND_TYPE**

Type: A derived data type defined in the **IEEE_ARITHMETIC** module that identifies a particular rounding mode. The values for **IEEE_ROUND_TYPE** must be one of the following named constants as defined in the **IEEE_ARITHMETIC** module:

**IEEE_NEAREST**

Rounds the exact result to the nearest representable value.

**IEEE_TO_ZERO**

Rounds the exact result to the next representable value, towards zero.
**IEEE_UP**
Rounds the exact result to the next representable value, towards positive infinity.

**IEEE_DOWN**
Rounds the exact result to the next representable value, towards negative infinity.

**IEEE_OTHER**
Indicates that the rounding mode does not conform to the IEEE standard.

### IEEE_FEATURES_TYPE

**Type:** A derived data type defined in the IEEE_FEATURES module that identifies the IEEE features to use. The values for IEEE_FEATURES_TYPE must be one of the following named constants as defined in the IEEE_FEATURES module:

<table>
<thead>
<tr>
<th>IEEE_DATATYPE</th>
<th>IEEE_DATATYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE_DENORMAL</td>
<td>IEEE_INVALID_FLAG</td>
</tr>
<tr>
<td>IEEE_DIVIDE</td>
<td>IEEE_NAN</td>
</tr>
<tr>
<td>IEEE_HALTING</td>
<td>IEEE_ROUNDING</td>
</tr>
<tr>
<td>IEEE_INEXACT_FLAG</td>
<td>IEEE_SQRT</td>
</tr>
<tr>
<td>IEEE_INF</td>
<td>IEEE_UNDERFLOW_FLAG</td>
</tr>
</tbody>
</table>

### IEEE Operators

The IEEE_ARITHMETIC module defines two sets of elemental operators for comparing variables of IEEE_CLASS_TYPE or IEEE_ROUND_TYPE.

**==** Allows you to compare two IEEE_CLASS_TYPE or two IEEE_ROUND_TYPE values. The operator returns true if the values are identical or false if they differ.

**/=** Allows you to compare two IEEE_CLASS_TYPE or two IEEE_ROUND_TYPE values. The operator returns true if the values differ or false if they are identical.

### IEEE PROCEDURES

To use the following IEEE procedures, you must add a USE IEEE_ARITHMETIC, USE IEEE_EXCEPTIONS, or USE IEEE_FEATURES statement to your source file as required. For more information on the USE statement, see “USE” on page 414.

#### Rules for Using IEEE Procedures

**Type:** XL Fortran supports all the named constants in the IEEE_FEATURES module.

The IEEE_ARITHMETIC module behaves as if it contained a USE statement for IEEE_EXCEPTIONS. All values that are public in IEEE_EXCEPTIONS remain public in IEEE_ARITHMETIC.

When the IEEE_EXCEPTIONS or the IEEE_ARITHMETIC modules are accessible, IEEE_OVERFLOW and IEEE_DIVIDE_BY_ZERO are supported in the scoping unit for all kinds of real and complex data. To determine the other exceptions supported use the IEEE_SUPPORT_FLAG function. Use IEEE_SUPPORT_HALTING to determine if halting is supported. Support of other
exceptions is influenced by the accessibility of the named constants

IEEE_INEXACT_FLAG, IEEE_INVALID_FLAG, and IEEE_UNDERFLOW_FLAG

of the IEEE_FEATURES module as follows:

- If a scoping unit has access to IEEE_UNDERFLOW_FLAG of IEEE_FEATURES,
  the scoping unit supports underflow and returns true from
  IEEE_SUPPORT_FLAG(IEEE_UNDERFLOW, X), for REAL(4) and REAL(8).
- If IEEE_INEXACT_FLAG or IEEE_INVALID_FLAG is accessible, the scoping
  unit supports the exception and returns true from the corresponding inquiry for
  REAL(4) and REAL(8).
- If IEEE_HALTING is accessible, the scoping unit supports halting control and
  returns true from IEEE_SUPPORT_HALTING(FLAG) for the flag.

If an exception flag signals on entry to a scoping unit that does not access
IEEE_EXCEPTIONS or IEEE_ARITHMETIC, the compiler ensures that the
exception flag is signaling on exit. If a flag is quiet on entry to such a scoping unit,
its signaling state remains unchanged on exit.

Further IEEE support is available through the IEEE_ARITHMETIC module.
Support is influenced by the accessibility of named constants in the
IEEE_FEATURES module:

- If a scoping unit has access to IEEE_DATATYPE of IEEE_FEATURES, the
  scoping unit supports IEEE arithmetic and returns true from
  IEEE_SUPPORT_DATATYPE(X) for REAL(4) and REAL(8).
- If IEEE_DENORMAL, IEEE_DIVIDE, IEEE_INF, IEEE_NAN,
  IEEE_ROUNDING, or IEEE_SQRT is accessible, the scoping unit supports the
  feature and returns true from the corresponding inquiry function for REAL(4)
  and REAL(8).
- For IEEE_ROUNDING, the scoping unit returns true for all the rounding modes
  IEEE_NEAREST, IEEE_TO_ZERO, IEEE_UP, and IEEE_DOWN for REAL(4)
  and REAL(8).

If the IEEE_EXCEPTIONS or IEEE_ARITHMETIC modules are accessed, and
IEEE_FEATURES is not, the supported subset of features is the same as if
IEEE_FEATURES was accessed.

**IEEE_CLASS(X)**

*Type:* An elemental IEEE class function. Returns the IEEE class of a floating-point
value.

*Module:* IEEE_ARITHMETIC

*Syntax:* Where X is of type real.

*Result type and attributes:* The result is of type IEEE_CLASS_TYPE.

*Rules:* To ensure compliance with the Fortran 2003 standard, the
**IEEE_SUPPORT_DATATYPE(X)** function must return with a value of true. If you
specify a data type of REAL(16), then IEEE_SUPPORT_DATATYPE will return
false, though the appropriate class type will still be returned.

*Examples:*
USE, INTRINSIC :: IEEE_ARITHMETIC
TYPE(IEEE_CLASS_TYPE) :: C
REAL :: X = -1.0
IF (IEEE_SUPPORT_DATATYPE(X)) THEN
    C = IEEE_CLASS(X)  ! C has class IEEE_NEGATIVE_NORMAL
ENDIF

IEEE_COPY_SIGN(X, Y)

Type: An elemental IEEE copy sign function. Returns the value of X with the sign of Y.

Module: IEEE_ARITHMETIC

Syntax: Where X and Y are of type real, though they may be of different kinds.

Result type and attributes: The result is of the same kind and type as X.

Rules: To ensure compliance with the Fortran 2003 standard, the IEEE_SUPPORT_DATATYPE(X) and IEEE_SUPPORT_DATATYPE(Y) must return with a value of true.

For supported IEEE special values, such as NaN and infinity, IEEE_COPY_SIGN returns the value of X with the sign of Y.

IEEE_COPY_SIGN ignores the -qif90=nosignedzero compiler option.

Note: XL Fortran REAL(16) numbers have no signed zero.

Examples: Example 1:
USE, INTRINSIC :: IEEE_ARITHMETIC
REAL :: X
DOUBLE PRECISION :: Y
X = 3.0
Y = -2.0
IF (IEEE_SUPPORT_DATATYPE(X) .AND. IEEE_SUPPORT_DATATYPE(Y)) THEN
    X = IEEE_COPY_SIGN(X,Y)  ! X has value -3.0
ENDIF

Example 2:
USE, INTRINSIC :: IEEE_ARITHMETIC
REAL :: X, Y
Y = 1.0
IF (IEEE_SUPPORT_DATATYPE(X)) THEN
    X = IEEE_VALUE(X, IEEE_NEGATIVE_INF)  ! X has value -inf
    X = IEEE_COPY_SIGN(X,Y)  ! X has value +inf
ENDIF

IEEE_GET_FLAG(FLAG, FLAG_VALUE)

Type: An elemental IEEE subroutine. Retrieves the status of the exception flag specified. Sets FLAG_VALUE to true if the flag is signaling, or false otherwise.

Module: IEEE_ARITHMETIC

Syntax: Where FLAG is an INTENT(IN) argument of type IEEE_FLAG_TYPE specifying the IEEE flag to obtain. FLAG_VALUE is an INTENT(OUT) default logical argument that contains the value of FLAG.

Examples:
USE, INTRINSIC :: IEEE_EXCEPTIONS
LOGICAL :: FLAG_VALUE
CALL IEEE_GET_FLAG(IEEE_OVERFLOW, FLAG_VALUE)
IF (FLAG_VALUE) THEN
   PRINT *, "Overflow flag is signaling."
ELSE
   PRINT *, "Overflow flag is quiet."
ENDIF

IEEE_GET_HALTING_MODE(FLAG, HALTING)

Type: An elemental IEEE subroutine. Retrieves the halting mode for an exception and sets HALTING to true if the exception specified by the flag will cause halting.

Module: IEEE_ARITHMETIC

Syntax: Where FLAG is an INTENT(IN) argument of type IEEE_FLAG_TYPE specifying the IEEE flag. HALTING is an INTENT(OUT) default logical.

Examples:
USE, INTRINSIC :: IEEE_EXCEPTIONS
LOGICAL HALTING
CALL IEEE_GET_HALTING_MODE(IEEE_OVERFLOW, HALTING)
IF (HALTING) THEN
   PRINT *, "The program will halt on an overflow exception."
ENDIF

IEEE_GET_ROUNDING_MODE (ROUND_VALUE)

Type: An IEEE subroutine. Sets ROUND_VALUE to the current IEEE rounding mode.

Module: IEEE_ARITHMETIC

Syntax: Where ROUND_VALUE is an INTENT(OUT) scalar of type IEEE_ROUND_TYPE.

Examples:
USE, INTRINSIC :: IEEE_ARITHMETIC
TYPE(IEEE_ROUND_TYPE) ROUND_VALUE
CALL IEEE_GET_ROUNDING_MODE(ROUND_VALUE) ! Store the rounding mode
IF (ROUND_VALUE == IEEE_OTHER) THEN
   PRINT *, "You are not using an IEEE rounding mode."
ENDIF

IEEE_GET_STATUS(STATUS_VALUE)

Type: An IEEE subroutine. Retrieves the current IEEE floating-point status.

Module: IEEE_ARITHMETIC

Syntax: Where STATUS_VALUE is an INTENT(OUT) scalar of type IEEE_STATUS_TYPE.

Rules: You can only use STATUS_VALUE in an IEEE_SET_STATUS invocation.

Examples:
USE, INTRINSIC :: IEEE_ARITHMETIC
TYPE(IEEE_STATUS_TYPE) STATUS_VALUE
...
CALL IEEE_GET_STATUS(STATUS_VALUE) ! Get status of all exception flags
CALL IEEE_SET_FLAG(IEEE_ALL, .FALSE.) ! Set all exception flags to quiet
... ! calculation involving exception handling
CALL IEEE_SET_STATUS(STATUS_VALUE) ! Restore the flags

IEEE_IS_FINITE(X)

Type: An elemental IEEE function. Tests whether a value is finite. Returns true if IEEE_CLASS(X) has one of the following values:

- IEEE_NEGATIVE_NORMAL
- IEEE_NEGATIVE_DENORMAL
- IEEE_NEGATIVE_ZERO
- IEEE_POSITIVE_ZERO
- IEEE_POSITIVE_DENORMAL
- IEEE_POSITIVE_NORMAL

It returns false otherwise.

Module: IEEE_ARITHMETIC

Syntax: Where X is of type real.

Result type and attributes: Where the result is of type default logical.

Rules: To ensure compliance with the Fortran 2003 standard, the IEEE_SUPPORT_DATATYPE(X) must return with a value of true.

Examples:

USE, INTRINSIC :: IEEE_ARITHMETIC
REAL :: X = 1.0
IF (IEEE_SUPPORT_DATATYPE(X)) THEN
  PRINT *, IEEE_IS_FINITE(X) ! Prints true
ENDIF

IEEE_IS_NAN(X)

Type: An elemental IEEE function. Tests whether a value is IEEE Not-a-Number. Returns true if IEEE_CLASS(X) has the value IEEE_SIGNALING_NAN or IEEE_QUIET_NAN. It returns false otherwise.

Module: IEEE_ARITHMETIC

Syntax: Where X is of type real.

Result type and attributes: Where the result is of type default logical.

Rules: To ensure compliance with the Fortran 2003 standard, the IEEE_SUPPORT_DATATYPE(X) and IEEE_SUPPORT_NAN(X) must return with a value of true.

Examples: Example 1:

USE, INTRINSIC :: IEEE_ARITHMETIC
REAL :: X = -1.0
IF (IEEE_SUPPORT_DATATYPE(X)) THEN
  IF (IEEE_SUPPORT_SQRT(X)) THEN ! IEEE-compliant SQRT function
    IF (IEEE_SUPPORT_NAN(X)) THEN
      PRINT *, IEEE_IS_NAN(SQRT(X)) ! Prints true
    ENDIF
  ENDIF
ENDIF
ENDIF
Example 2:

```fortran
USE, INTRINSIC :: IEEE_ARITHMETIC
REAL :: X = -1.0
IF (IEEE_SUPPORT_STANDARD(X)) THEN
    PRINT *, IEEE_IS_NAN(SQRT(X))  ! Prints true
ENDIF
```

**IEEE_IS_NEGATIVE(X)**

**Type:** An elemental IEEE function. Tests whether a value is negative. Returns true if `IEEE_CLASS(X)` has one of the following values:
- `IEEE_NEGATIVE_NORMAL`
- `IEEE_NEGATIVE_DENORMAL`
- `IEEE_NEGATIVE_ZERO`
- `IEEE_NEGATIVE_INF`

It returns false otherwise.

**Module:** IEEE_ARITHMETIC

**Syntax:** Where `X` is of type real.

**Result type and attributes:** Where the result is of type default logical.

**Rules:** To ensure compliance with the Fortran 2003 standard, the `IEEE_SUPPORT_DATATYPE(X)` must return with a value of true.

**Examples:**

```fortran
USE, INTRINSIC :: IEEE_ARITHMETIC
IF (IEEE_SUPPORT_DATATYPE(X)) THEN
    PRINT *, IEEE_IS_NEGATIVE(X)  ! Prints false
ENDIF
```

**IEEE_IS_NORMAL(X)**

**Type:** An elemental IEEE function. Tests whether a value is normal. Returns true if `IEEE_CLASS(X)` has one of the following values:
- `IEEE_NEGATIVE_NORMAL`
- `IEEE_NEGATIVE_ZERO`
- `IEEE_POSITIVE_ZERO`
- `IEEE_POSITIVE_NORMAL`

It returns false otherwise.

**Module:** IEEE_ARITHMETIC

**Syntax:** Where `X` is of type real.

**Result type and attributes:** Where the result is of type default logical.

**Rules:** To ensure compliance with the Fortran 2003 standard, the `IEEE_SUPPORT_DATATYPE(X)` must return with a value of true.

**Examples:**

```fortran
USE, INTRINSIC :: IEEE_ARITHMETIC
REAL :: X = -1.0
IF (IEEE_SUPPORT_DATATYPE(X)) THEN
    PRINT *, IEEE_IS_NORMAL(X)  ! Prints false
ENDIF
```
IF (IEEE_SUPPORT_SQRT(X)) THEN ! IEEE-compliant SQRT function
    PRINT *, IEEE_IS_NORMAL(SQRT(X)) ! Prints false
ENDIF
ENDIF

**IEEE_LOGB(X)**

**Type:** An elemental IEEE function. Returns unbiased exponent in the IEEE floating-point format. If the value of X is neither zero, infinity, or NaN, the result has the value of the unbiased exponent of X, equal to EXPONENT(X)–1.

**Module:** IEEE_ARITHMETIC

**Syntax:** Where X is of type real.

**Result type and attributes:** Where the result is the same type and kind as X.

**Rules:** To ensure compliance with the Fortran 2003 standard, the IEEE_SUPPORT_DATATYPE(X) must return with a value of true.

If X is zero, the result is negative infinity.

If X is infinite, the result is positive infinity.

If X is NaN, the result is nan.

**Examples:**

USE, INTRINSIC :: IEEE_ARITHMETIC
IF (IEEE_SUPPORT_DATATYPE(1.1)) THEN
    PRINT *, IEEE_LOGB(1.1) ! Prints 0.0
ENDIF

**IEEE_NEXT_AFTER(X, Y)**

**Type:** An elemental IEEE function. Returns the next machine-representable neighbor of X in the direction towards Y.

**Module:** IEEE_ARITHMETIC

**Syntax:** Where X and Y are of type real.

**Result type and attributes:** Where the result is the same type and kind as X.

**Rules:** To ensure compliance with the Fortran 2003 standard, the IEEE_SUPPORT_DATATYPE(X) and IEEE_SUPPORT_DATATYPE(Y) must return with a value of true.

If X and Y are equal the function returns X without signaling an exception. If X and Y are not equal, the function returns the next machine-representable neighbor of X in the direction towards Y.

The neighbors of zero, of either sign, are both nonzero.

IEEE_OVERFLOW and IEEE_INEXACT are signaled when X is finite but IEEE_NEXT_AFTER(X, Y) is infinite.

IEEE_UNDERFLOW and IEEE_INEXACT are signaled when IEEE_NEXT_AFTER(X, Y) is denormalized or zero.
If \( X \) or \( Y \) is a quiet NaN, the result is one of the input NaN values.

**Examples:** Example 1:
```
USE, INTRINSIC :: IEEE_ARITHMETIC
REAL :: X = 1.0, Y = 2.0
IF (IEEE_SUPPORT_DATATYPE(X)) THEN
  PRINT *, (IEEE_NEXT_AFTER(X,Y) == X + EPSILON(X)) ! Prints true
ENDIF
```

Example 2:
```
USE, INTRINSIC :: IEEE_ARITHMETIC
REAL(4) :: X = 0.0, Y = 1.0
IF (IEEE_SUPPORT_DATATYPE(X)) THEN
  PRINT *, (IEEE_NEXT_AFTER(X,Y) == 2.0**(-149)) ! Prints true
ENDIF
```

**IEEE Rem(\( X, Y \))**

**Type:** An elemental IEEE remainder function. The result value, regardless of the rounding mode, is exactly \( X-Y*N \), where \( N \) is the integer nearest to the exact value \( X/Y \); whenever \( |N - X/Y| = 1/2 \), \( N \) is even.

**Module:** IEEE_ARITHMETIC

**Syntax:** Where \( X \) and \( Y \) are of type real.

**Result type and attributes:** Where the result is of type real with the same kind as the argument with greater precision.

**Rules:** To ensure compliance with the Fortran 2003 standard, the IEEE_SUPPORT_DATATYPE(\( X \)) and IEEE_SUPPORT_DATATYPE(\( Y \)) must return with a value of true.

If the result value is zero, the sign is the same as \( X \).

**Examples:**
```
USE, INTRINSIC :: IEEE_ARITHMETIC
IF (IEEE_SUPPORT_DATATYPE(4.0)) THEN
  PRINT *, IEEE_REM(4.0,3.0) ! Prints 1.0
  PRINT *, IEEE_REM(3.0,2.0) ! Prints -1.0
  PRINT *, IEEE_REM(5.0,2.0) ! Prints 1.0
ENDIF
```

**IEEE_Rint(\( X \))**

**Type:** An elemental IEEE function. Rounds to an integer value according to the current rounding mode.

**Module:** IEEE_ARITHMETIC

**Syntax:** Where \( X \) is of type real.

**Result type and attributes:** Where the result is the same type and kind as \( X \).

**Rules:** To ensure compliance with the Fortran 2003 standard, the IEEE_SUPPORT_DATATYPE(\( X \)) must return with a value of true.

If the result has the value zero, the sign is that of \( X \).
Examples:

USE, INTRINSIC :: IEEE_ARITHMETIC
IF (IEEE_SUPPORT_DATATYPE(1.1)) THEN
   CALL IEEE_SET_ROUNDING_MODE(IEEE_NEAREST)
   PRINT *, IEEE_RINT(1.1) ! Prints 1.0
   CALL IEEE_SET_ROUNDING_MODE(IEEE_UP)
   PRINT *, IEEE_RINT(1.1) ! Prints 2.0
ENDIF

IEEE_SCALB(X, I)

Type: An elemental IEEE function. Returns X * 2^I.

Module: IEEE_ARITHMETIC

Syntax: Where X is of type real and I is of type INTEGER.

Result type and attributes: Where the result is the same type and kind as X.

Rules: To ensure compliance with the Fortran 2003 standard, the
IEEE_SUPPORT_DATATYPE(X) must return with a value of true.

If X * 2^I is representable as a normal number, then the result is a normal number.

If X is finite and X * 2^I is too large the IEEE_OVERFLOW exception occurs. The result value is infinity with the sign of X.

If X * 2^I is too small and there is a loss of accuracy, the IEEE_UNDERFLOW exception occurs. The result is the nearest representable number with the sign of X.

If X is infinite, the result is the same as X with no exception signals.

Examples:

USE, INTRINSIC :: IEEE_ARITHMETIC
IF (IEEE_SUPPORT_DATATYPE(1.0)) THEN
   PRINT *, IEEE_SCALB(1.0, 2) ! Prints 4.0
ENDIF

IEEE_SELECTED_REAL_KIND([P, R])

Type: A transformational IEEE function. Returns a value of the kind type parameter of an IEEE real data type with decimal precision of at least P digits, and a decimal exponent range of at least R.

Module: IEEE_ARITHMETIC

Syntax: Where P and R are both scalar optional arguments of type integer.

Rules: If the kind type parameter is not available and the precision is not available, the result is −1. If the kind type parameter is not available and the exponent range is not available, the result is −2. If the kind type parameter is not available and if neither the precision or the exponent range is available, the result is −3.

If more than one kind type parameter value is applicable, the value returned is the one with the smallest decimal precision. If there are several values, the smallest of these kind values is returned.

Examples:
USE, INTRINSIC :: IEEE_ARITHMETIC

! P and R fit in a real(4)
PRINT *, IEEE_SELECTED_REAL_KIND(6,37) ! prints 4

! P needs at least a real(8)
PRINT *, IEEE_SELECTED_REAL_KIND(14,37) ! prints 8
! R needs at least a real(8)
PRINT *, IEEE_SELECTED_REAL_KIND(6,307) ! prints 8

! P is too large
PRINT *, IEEE_SELECTED_REAL_KIND(40,37) ! prints -1
! R is too large
PRINT *, IEEE_SELECTED_REAL_KIND(6,400) ! prints -2
! P and R are both too large
PRINT *, IEEE_SELECTED_REAL_KIND(40,400) ! prints -3

END

IEEE_SET_FLAG(FLAG, FLAG_VALUE)

Type: An IEEE subroutine. Assigns a value to an IEEE exception flag.

Module: IEEE_EXCEPTIONS

Syntax: Where FLAG is an INTENT(IN) scalar or array argument of type
IEEE_FLAG_TYPE corresponding to the value of the flag to be set. FLAG_VALUE
is an INTENT(IN) scalar or array argument of type logical, corresponding to
the desired status of the exception flag. The value of FLAG_VALUE should be
conformable with the value of FLAG.

Rules: If FLAG_VALUE is true, the exception flag specified by FLAG is set to
signaling. Otherwise, the flag is set to quiet.

Each element of FLAG must have a unique value.

Examples:
USE, INTRINSIC :: IEEE_EXCEPTIONS
CALL IEEE_SET_FLAG(IEEE_OVERFLOW, .TRUE.)
! IEEE_OVERFLOW is now signaling

IEEE_SET_HALTING_MODE(FLAG, HALTING)

Type: An IEEE subroutine. Controls continuation or halting after an exception.

Module: IEEE_EXCEPTIONS

Syntax: Where FLAG is an INTENT(IN) scalar or array argument of type
IEEE_FLAG_TYPE corresponding to the exception flag for which holding applies.
HALTING is an INTENT(IN) scalar or array argument of type logical,
corresponding to the desired halting status. By default exceptions will not cause
halting in XL Fortran. The value of HALTING should be conformable with the
value of FLAG.

Rules: To ensure compliance with the Fortran 2003 standard, the
IEEE_SUPPORT_DATATYPE(X) must return with a value of true.

If HALTING is true, the exception specified by FLAG will cause halting. Otherwise,
execution will continue after the exception.
Each element of \textit{FLAG} must have a unique value.

\textbf{Examples:}

\begin{verbatim}
   @PROCESS FLOAT(NOFOLD)
   USE, INTRINSIC :: IEEE_EXCEPTIONS
   REAL :: X
   CALL IEEE_SET_HALTING_MODE(IEEE_DIVIDE_BY_ZERO, .TRUE.)
   X = 1.0 / 0.0
   ! Program will halt with a divide-by-zero exception
\end{verbatim}

\textbf{IEEE\_SET\_ROUNDING\_MODE (ROUND\_VALUE)}

\textbf{Type:} An IEEE subroutine. Sets the current rounding mode.

\textbf{Module:} IEEE\_ARITHMETIC

\textbf{Syntax:} Where \textit{ROUND\_VALUE} is an \texttt{INTENT(IN)} argument of type \texttt{IEEE\_ROUND\_TYPE} specifying the rounding mode.

\textbf{Rules:} To ensure compliance with the Fortran 2003 standard, the \texttt{IEEE\_SUPPORT\_DATATYPE(X)} and \texttt{IEEE\_SUPPORT\_ROUNDING (ROUND\_VALUE, X)} must return with a value of true.

The compilation unit calling this program must be compiled with the \texttt{-qfloat=rrm} compiler option.

All compilation units calling programs compiled with the \texttt{-qfloat=rrm} compiler option must also be compiled with this option.

\textbf{Examples:}

\begin{verbatim}
   USE, INTRINSIC :: IEEE\_ARITHMETIC
   IF (IEEE\_SUPPORT\_DATATYPE(1.1)) THEN
      CALL IEEE\_SET\_ROUNDING\_MODE(IEEE\_NEAREST)
      PRINT *, IEEE\_RINT(1.1) ! Prints 1.0
      CALL IEEE\_SET\_ROUNDING\_MODE(IEEE\_UP)
      PRINT *, IEEE\_RINT(1.1) ! Prints 2.0
   ENDIF
\end{verbatim}

\textbf{IEEE\_SET\_STATUS(STATUS\_VALUE)}

\textbf{Type:} An IEEE subroutine. Restores the value of the floating-point status.

\textbf{Module:} IEEE\_ARITHMETIC

\textbf{Syntax:} Where \textit{STATUS\_VALUE} is an \texttt{INTENT(IN)} argument of type \texttt{IEEE\_STATUS\_TYPE} specifying the floating-point status.

\textbf{Rules:} \textit{STATUS\_VALUE} must have been set previously by \texttt{IEEE\_GET\_STATUS}.

\textbf{IEEE\_SUPPORT\_DATATYPE or IEEE\_SUPPORT\_DATATYPE(X)}

\textbf{Type:} An inquiry IEEE function. Determines whether the current implementation supports IEEE arithmetic. Support means using an IEEE data format and performing the binary operations of +, -, and \* as in the IEEE standard whenever the operands and result all have normal values.

\textbf{Note:} NaN and Infinity are not fully supported for \texttt{REAL(16)}. Arithmetic operations do not necessarily propagate these values.
Module: IEEE_ARITHMETIC

Syntax: Where X is a scalar or array valued argument of type real.

Result type and attributes: The result is a scalar of type default logical.

Rules: If X is absent, the function returns a value of false.

If X is present and REAL(16), the function returns a value of false. Otherwise the function returns true.

Examples:
USE, INTRINSIC :: IEEE_ARITHMETIC
TYPE(IEEE_STATUS_TYPE) STATUS_VALUE
...
CALL IEEE_GET_STATUS(STATUS_VALUE)  ! Get status of all exception flags
CALL IEEE_SET_FLAG(IEEE_ALL,.FALSE.) ! Set all exception flags to quiet
... ! calculation involving exception handling
CALL IEEE_SET_STATUS(STATUS_VALUE) ! Restore the flags

IEEE_SUPPORT_DENORMAL or IEEE_SUPPORT_DENORMAL(X)

Type: An inquiry IEEE function. Determines whether the current implementation supports denormalized numbers.

Module: IEEE_ARITHMETIC

Syntax: Where X is a scalar or array valued argument of type real.

Result type and attributes: The result is a scalar of type default logical.

Rules: To ensure compliance with the Fortran 2003 standard, the IEEE_SUPPORT_DATATYPE(X) must return with a value of true.

The result has a value of true if the implementation supports arithmetic operations and assignments with denormalized numbers for all arguments of type real where X is absent, or for real variables of the same kind type parameter as X. Otherwise, the result has a value of false.

IEEE_SUPPORT_DIVIDE or IEEE_SUPPORT_DIVIDE(X)

Type: An inquiry IEEE function. Determines whether the current implementation supports division to the accuracy of the IEEE standard.

Module: IEEE_ARITHMETIC

Syntax: Where X is a scalar or array valued argument of type real.

Result type and attributes: The result is a scalar of type default logical.

Rules: To ensure compliance with the Fortran 2003 standard, the IEEE_SUPPORT_DATATYPE(X) must return with a value of true.

The result has a value of true if the implementation supports division with the accuracy specified by the IEEE standard for all arguments of type real where X is absent, or for real variables of the same kind type parameter as X. Otherwise, the result has a value of false.
IEEE_SUPPORT_FLAG(FLAG) or IEEE_SUPPORT_FLAG(FLAG, X)

Type: An inquiry IEEE function. Determines whether the current implementation supports an exception.

Module: IEEE_EXCEPTIONS

Syntax: Where FLAG is a scalar argument of IEEE_FLAG_TYPE. X is a scalar or array valued argument of type real.

Result type and attributes: The result is a scalar of type default logical.

Rules: The result has a value of true if the implementation supports detection of the exception specified for all arguments of type real where X is absent, or for real variables of the same kind type parameter as X. Otherwise, the result has a value of false.

If X is absent, the result has a value of false.

If X is present and of type REAL(16), the result has a value of false. Otherwise the result has a value of true.

IEEE_SUPPORT_HALTING(FLAG)

Type: An inquiry IEEE function. Determines whether the current implementation supports the ability to abort or continue execution after an exception occurs. Support by the current implementation includes the ability to change the halting mode using IEEE_SET_HALTING(FLAG).

Module: IEEE_EXCEPTIONS

Syntax: Where FLAG is an INTENT(IN) argument of IEEE_FLAG_TYPE.

Result type and attributes: The result is a scalar of type default logical.

Rules: The result returns with a value of true for all flags.

IEEE_SUPPORT_INF or IEEE_SUPPORT_INF(X)

Type: An inquiry IEEE function. Support indicates that IEEE infinity behavior for unary and binary operations, including those defined by intrinsic functions and by functions in intrinsic modules, complies with the IEEE standard.

Module: IEEE_ARITHMETIC

Syntax: Where X is a scalar or array valued argument of type real.

Result type and attributes: The result is a scalar of type default logical.

Rules: To ensure compliance with the Fortran 2003 standard, the IEEE_SUPPORT_DATATYPE(X) must return with a value of true.

The result has a value of true if the implementation supports IEEE positive and negative infinities for all arguments of type real where X is absent, or for real variables of the same kind type parameter as X. Otherwise, the result has a value of false.
If \( X \) is of type \texttt{REAL(16)}, the result has a value of false. Otherwise the result has a value of true.

**IEEE_SUPPORT_IO or IEEE_SUPPORT_IO(X)**

**Type:** An inquiry IEEE function. Determines whether the current implementation supports IEEE base conversion rounding during formatted input/output. Support refers to the ability to do IEEE base conversion during formatted input/output as described in the IEEE standard for the modes \texttt{IEEE_UP}, \texttt{IEEE_DOWN}, \texttt{IEEE_ZERO}, and \texttt{IEEE_NEAREST} for all arguments of type real where \( X \) is absent, or for real variables of the same kind type parameter as \( X \).

**Module:** \texttt{IEEE_ARITHMETIC}

**Syntax:** Where \( X \) is a scalar or array valued argument of type real.

**Result type and attributes:** The result is a scalar of type default logical.

**Rules:** To ensure compliance with the Fortran 2003 standard, the \texttt{IEEE_SUPPORT_DATATYPE(X)} must return with a value of true.

If \( X \) is present and of type \texttt{REAL(16)}, the result has a value of false. Otherwise, the result returns a value of true.

**IEEE_SUPPORT_NAN or IEEE_SUPPORT_NAN(X)**

**Type:** An inquiry IEEE function. Determines whether the current implementation supports the IEEE Not-a-Number facility. Support indicates that IEEE NaN behavior for unary and binary operations, including those defined by intrinsic functions and by functions in intrinsic modules, conforms to the IEEE standard.

**Module:** \texttt{IEEE_ARITHMETIC}

**Syntax:** Where \( X \) is a scalar or array valued argument of type real.

**Result type and attributes:** The result is a scalar of type default logical.

**Rules:** To ensure compliance with the Fortran 2003 standard, the \texttt{IEEE_SUPPORT_DATATYPE(X)} must return with a value of true.

If \( X \) is absent, the result has a value of false.

If \( X \) is present and of type \texttt{REAL(16)}, the result has a value of false. Otherwise the result returns a value of true.

**IEEE_SUPPORT_ROUNDING (ROUND_VALUE) or IEEE_SUPPORT_ROUNDING (ROUND_VALUE, X)**

**Type:** An inquiry IEEE function. Determines whether the current implementation supports a particular rounding mode for arguments of type real. Support indicates the ability to change the rounding mode using \texttt{IEEE_SET_ROUNDING_MODE}.

**Module:** \texttt{IEEE_ARITHMETIC}

**Syntax:** Where \texttt{ROUND_VALUE} is a scalar argument of \texttt{IEEE_ROUND_TYPE}. \( X \) is a scalar or array valued argument of type real.
Result type and attributes: The result is a scalar of type default logical.

Rules: To ensure compliance with the Fortran 2003 standard, the IEEE_SUPPORT_DATATYPE(X) must return with a value of true.

If X is absent, the result has a value of true if the implementation supports the rounding mode defined by ROUND_VALUE for all arguments of type real. Otherwise, it has a value of false.

If X is present, the result returns a value of true if the implementation supports the rounding mode defined by ROUND_VALUE for real variables of the same kind type parameter as X. Otherwise, the result has a value of false.

If X is present and of type REAL(16), the result returns a value of false when ROUND_VALUE has a value of IEEE_NEAREST. Otherwise the result returns a value of true.

If ROUND_VALUE has a value of IEEE_OTHER the result has a value of false.

IEEE_SUPPORT_SQRT or IEEE_SUPPORT_SQRT(X)

Type: An inquiry IEEE function. Determines whether the current implementation supports the SQRT as defined by the IEEE standard.

Module: IEEE_ARITHMETIC

Syntax: Where X is a scalar or array valued argument of type real.

Result type and attributes: The result is a scalar of type default logical.

Rules: To ensure compliance with the Fortran 2003 standard, the IEEE_SUPPORT_DATATYPE(X) must return with a value of true.

If X is absent, the result returns a value of true if SQRT adheres to IEEE conventions for all variables of type REAL. Otherwise, the result has a value of false.

If X is present, the result returns a value of true if SQRT adheres to IEEE conventions for all variables of type REAL with the same kind type parameter as X. Otherwise, the result has a value of false.

If X is present and of type REAL(16), the result has a value of false. Otherwise the result returns a value of true.

IEEE_SUPPORT_STANDARD or IEEE_SUPPORT_STANDARD(X)

Type: An inquiry IEEE function. Determines whether all facilities defined in the Fortran 2003 standard are supported.

Module: IEEE_ARITHMETIC

Syntax: Where X is a scalar or array valued argument of type real.

Result type and attributes: The result is a scalar of type default logical.

Rules: If X is absent, the result returns a value of false since XL Fortran supports REAL(16).
If $X$ is present, the result returns a value of true if the following functions also return true:

- `IEEE_SUPPORT_DATATYPE(X)`
- `IEEE_SUPPORT_DENORMAL(X)`
- `IEEE_SUPPORT_DIVIDE(X)`
- `IEEE_SUPPORT_FLAG(FLAG, X)` for every valid flag.
- `IEEE_SUPPORT_HALTING(FLAG)` for every valid flag.
- `IEEE_SUPPORT_INF(X)`
- `IEEE_SUPPORT_NAN(X)`
- `IEEE_SUPPORT_ROUNDING(ROUND_VALUE, X)` for every valid `ROUND_VALUE`
- `IEEE_SUPPORT_SQRT(X)`

Otherwise, the result returns a value of false.

**IEEE_UNORDERED(X, Y)**

Type: An elemental IEEE unordered function.

Module: `IEEE_ARITHMETIC`

Syntax: Where $X$ and $Y$ are of type real.

Result type and attributes: The result is of type default logical.

Rules: To ensure compliance with the Fortran 2003 standard, the `IEEE_SUPPORT_DATATYPE(X)` and `IEEE_SUPPORT_DATATYPE(Y)` must return with a value of true.

Unordered function that returns with a value of true if $X$ or $Y$ is a NaN. Otherwise the function returns with a value of false.

Examples:

```fortran
USE, INTRINSIC :: IEEE_ARITHMETIC
REAL X, Y
X = 0.0
Y = IEEE_VALUE(Y, IEEE_QUIET_NAN)
PRINT *, IEEE_UNORDERED(X,Y)  ! Prints true
END
```

**IEEE_VALUE(X, CLASS)**

Type: An elemental IEEE function. Generates an IEEE value as specified by `CLASS`.

Note: Implementation of this function is platform and compiler dependent due to variances in NaN processing on differing platforms. A NaN value saved in a binary file that is read on a different platform than the one that generated the value will have unspecified results.

Module: `IEEE_ARITHMETIC`

Syntax: Where $X$ is of type real. `CLASS` is of type `IEEE_CLASS_TYPE`.

Result type and attributes: The result is of the same type and kind as $X$. 

714  XL Fortran Language Reference
Rules: To ensure compliance with the Fortran 2003 standard, the IEEE_SUPPORT_DATATYPE(X) must return with a value of true.

IEEE_SUPPORT_NAN(X) must be true if the value of CLASS is IEEE_SIGNALING_NAN or IEEE_QUIET_NAN.

IEEE_SUPPORT_INF(X) must be true if the value of CLASS is IEEE_NEGATIVE_INF or IEEE_POSITIVE_INF.

IEEE_SUPPORT_DENORMAL(X) must be true if the value of CLASS is IEEE_NEGATIVE_DENORMAL or IEEE_POSITIVE_DENORMAL.

Multiple calls of IEEE_VALUE(X, CLASS) return the same result for a particular value of X, if kind type parameter and CLASS remain the same.

If a compilation unit calls this program with a CLASS value of IEEE_SIGNALING_NAN, the compilation unit must be compiled with the –qfloat=nans compiler option.

Examples:
USE, INTRINSIC :: IEEE_ARITHMETIC
REAL :: X
IF (IEEE_SUPPORT_DATATYPE(X)) THEN
  X = IEEE_VALUE(X, IEEE_NEGATIVE_INF)
  PRINT *, 'X ! Prints -inf'
END IF

Rules for floating-point status

An exception flag set to signaling remains signaling until set to quiet by either the IEEE_SET_FLAG or IEEE_SET_STATUS subroutines.

The compiler ensures that a call from scoping units using the IEEE_EXCEPTIONS or IEEE_ARITHMETIC intrinsic modules does not change the floating-point status other than by setting exception flags to signaling.

If a flag is set to signaling on entry into a scoping unit that uses the IEEE_EXCEPTIONS or IEEE_ARITHMETIC modules, the flag is set to quiet and then restored to signaling when leaving that scoping unit.

In a scoping unit that uses the IEEE_EXCEPTIONS or IEEE_ARITHMETIC modules, the rounding and halting modes do not change on entry. On return, the rounding and halting modes are the same as on entry.

Evaluating a specification expression can cause an exception to signal.

Exception handlers must not use the IEEE_EXCEPTIONS or IEEE_ARITHMETIC modules.

The following rules apply to format processing and intrinsic procedures:
• The status of a signaling flag, either signaling or quiet, does not change because of an intermediate calculation that does not affect the result.
• If an intrinsic procedure executes normally, the values of the flags IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, and IEEE_INVALID remain the same on entry to the procedure.
• If a real or complex result is too large for the intrinsic to handle, 
  \texttt{IEEE\_OVERFLOW} may signal.
• If a real or complex result is a NaN because of an invalid operation, 
  \texttt{IEEE\_INVALID} may signal.

In a sequence of statements that has no invocations of \texttt{IEEE\_GET\_FLAG}, \texttt{IEEE\_SET\_FLAG}, \texttt{IEEE\_GET\_STATUS}, \texttt{IEEE\_SET\_HALTING}, or \texttt{IEEE\_SET\_STATUS}, the following applies. If the execution of an operation would cause an exception to signal but after execution of the sequence no value of a variable depends on the operation, whether the exception is signaling depends on the optimization level. Optimization transformations may eliminate some code, and thus IEEE exception flags signaled by the eliminated code will not signal.

An exception will not signal if this could arise only during execution of an operation beyond those required or permitted by the standard.

For procedures defined by means other than Fortran, it is the responsibility of the user to preserve floating-point status.

XL Fortran does not always detect floating-point exception conditions for extended precision values. If you turn on floating-point exception trapping in programs that use extended precision, XL Fortran may also generate signals in cases where an exception does not really occur. See Detecting and trapping floating-point exceptions in the XL Fortran Optimization and Programming Guide for more information.

Fortran 2003 IEEE derived types, constants, and operators are incompatible with the floating-point and inquiry procedures in \texttt{xlf\_fp\_util, fpsets, and fpgets} procedures. A value obtained from an IEEE procedure cannot be used in non-IEEE procedures. Within a single scoping unit, do not mix calls to the procedures in \texttt{xlf\_fp\_util, fpsets, and fpgets} with calls to the IEEE procedures. These procedures may change the floating-point status when called from scoping units that use the \texttt{IEEE\_EXCEPTIONS} or \texttt{IEEE\_ARITHMETIC} modules.

\section*{Examples}

\textbf{Example 1:} In the following example, the main program calls procedure \texttt{P} which uses the \texttt{IEEE\_ARITHMETIC} module. The procedure changes the floating-point status before returning. The example displays the changes to the floating-point status before calling procedure \texttt{P}, on entry into the procedure, on exit from \texttt{P}, and after returning from the procedure.

\begin{verbatim}
PROGRAM MAIN
    USE, INTRINSIC :: IEEE_ARITHMETIC

    INTERFACE
        SUBROUTINE P()
            USE IEEE_ARITHMETIC
        END SUBROUTINE P
    END INTERFACE

    LOGICAL, DIMENSION(5) :: FLAG_VALUES
    TYPE(IEEE\_ROUND\_TYPE) :: ROUND\_VALUE

    CALL IEEE\_SET\_FLAG(IEEE\_OVERFLOW, .TRUE.)
    CALL IEEE\_GET\_FLAG(IEEE\_ALL, FLAG\_VALUES)
    PRINT *, "MAIN: FLAGS ", FLAG\_VALUES
    CALL P()

END PROGRAM MAIN
\end{verbatim}
CALL IEEE_GET_FLAG(IEEE_ALL, FLAG_VALUES)
PRINT *, "MAIN: FLAGS ", FLAG_VALUES

CALL IEEE_GET_ROUNDING_MODE(ROUND_VALUE)
IF (ROUND_VALUE == IEEE_NEAREST) THEN
   PRINT *, "MAIN: Rounding mode: IEEE_NEAREST"
ENDIF
END PROGRAM MAIN

SUBROUTINE P()
USE IEEE_ARITHMETIC
LOGICAL, DIMENSION(5) :: FLAG_VALUES
TYPE(IEEE_ROUND_TYPE) :: ROUND_VALUE

CALL IEEE_GET_FLAG(IEEE_ALL, FLAG_VALUES)
PRINT *, "P: Flags on entry: ", FLAG_VALUES

CALL IEEE_SET_ROUNDING_MODE(IEEE_TO_ZERO)
CALL IEEE_SET_FLAG(IEEE_UNDERFLOW, .TRUE.)

CALL IEEE_GET_ROUNDING_MODE(ROUND_VALUE)
IF (ROUND_VALUE == IEEE_TO_ZERO) THEN
   PRINT *, "P: Rounding mode on exit: IEEE_TO_ZERO"
ENDIF
CALL IEEE_GET_FLAG(IEEE_ALL, FLAG_VALUES)
PRINT *, "P: Flags on exit: ", FLAG_VALUES
END SUBROUTINE P

When using the --qstrictieeeemod compiler option to ensure compliance with rules for IEEE arithmetic, exception flags set before calling P are cleared on entry to P. Changes to the floating-point status occurring in P are undone when P returns, with the exception that flags set in P remain set after P returns:

MAIN: FLAGS T F F F F
   P: FLAGS on entry: F F F F F
   P: Rounding mode on exit: IEEE_TO_ZERO
   P: Flags on exit: F F F T F
MAIN: FLAGS T F F T F
MAIN: Rounding mode: IEEE_NEAREST

When the --qnostrictieeeemod compiler option is in effect, exception flags which were set before calling P remain set on entry to P. Changes to the floating-point status occurring in P are propagated to the caller:

MAIN: FLAGS T F F F F
   P: Flags on entry: T F F F F
   P: Rounding mode on exit: IEEE_TO_ZERO
   P: Flags on exit: T F F T F
MAIN: FLAGS T F F T F

Example 2: In the following example, the main program calls procedure Q which uses neither IEEE_ARITHMETIC nor IEEE_EXCEPTIONS. Procedure Q changes the floating-point status before returning. The example displays the changes to the floating-point status before calling Q, on entry into the procedure, on exit from Q, and after returning from the procedure.

PROGRAM MAIN
USE, INTRINSIC :: IEEE_ARITHMETIC

LOGICAL, DIMENSION(5) :: FLAG_VALUES
TYPE(IEEE_ROUND_TYPE) :: ROUND_VALUE

CALL IEEE_SET_FLAG(IEEE_OVERFLOW, .TRUE.)
CALL IEEE_GET_FLAG(IEEE_ALL, FLAG_VALUES)
PRINT *, "MAIN: FLAGS ",FLAG_VALUES

CALL Q()

CALL IEEE_GET_FLAG(IEEE_ALL, FLAG_VALUES)
PRINT *, "MAIN: FLAGS ",FLAG_VALUES

CALL IEEE_GET_ROUNDING_MODE(ROUND_VALUE)
IF (ROUND_VALUE == IEEE_NEAREST) THEN
  PRINT *, "MAIN: ROUNDING MODE: IEEE_NEAREST"
ENDIF
END PROGRAM MAIN

SUBROUTINE Q()
USE XLF_FP_UTIL

FUNCTION GET_FLAGS()
  LOGICAL, DIMENSION(5) :: GET_FLAGS
END FUNCTION

LOGICAL, DIMENSION(5) :: FLAG_VALUES
INTEGER(FP_MODE_KIND) :: OLDMODE

FLAG_VALUES = GET_FLAGS()
PRINT *, " Q: FLAGS ON ENTRY: ", FLAG_VALUES

CALL CLR_FPSCR_FLAGS(FP_OVERFLOW)
OLDMODE = SET_ROUND_MODE(FP_RND_RZ)
CALL SET_FPSCR_FLAGS(TRP_OVERFLOW)
CALL SET_FPSCR_FLAGS(FP_UNDERFLOW)

IF (GET_ROUND_MODE() == FP_RND_RZ) THEN
  PRINT *, " Q: ROUNDING MODE ON EXIT: TO_ZERO"
ENDIF

FLAG_VALUES = GET_FLAGS()
PRINT *, " Q: FLAGS ON EXIT: ", FLAG_VALUES
END SUBROUTINE Q

! PRINT THE STATUS OF ALL EXCEPTION FLAGS
FUNCTION GET_FLAGS()
USE XLF_FP_UTIL
LOGICAL, DIMENSION(5) :: GET_FLAGS
INTEGER(FPSCR_KIND), DIMENSION(5) :: FLAGS
INTEGER I

FLAGS = (/ FP_OVERFLOW, FP_DIV_BY_ZERO, FP_INVALID, &
& FP_UNDERFLOW, FP_INEXACT /)
DO I=1,5
  GET_FLAGS(I) = (GET_FPSCR_FLAGS(FLAGS(I)) /= 0)
END DO
END FUNCTION

When using the –qstrictieemod compiler option to ensure compliance with rules for IEEE arithmetic, exception flags set before Q remain set on entry into Q. Changes to the floating-point status occurring in Q are undone when Q returns, with the exception that flags set in Q remain set after Q returns:

MAIN: FLAGS  T F F F F
Q: FLAGS ON ENTRY:  T F F F F
Q: ROUNDING MODE ON EXIT: TOZERO
Q: FLAGS ON EXIT:  F F F T F
MAIN: FLAGS  T F F T F
MAIN: ROUNDING MODE: IEEE_NEAREST
When the –qnostrictieemod option is in effect, exception flags set before calling Q remain set on entry into Q. Changes to the floating point status occurring in Q are propagated to the caller.

MAIN: FLAGS  T  F  F  F  F
  Q: FLAGS ON ENTRY:  T  F  F  F  F
  Q: ROUNDING MODE ON EXIT: TO_ZERO
  Q: FLAGS ON EXIT:  F  F  F  T  F
MAIN: FLAGS  F  F  T  F

End of Fortran 2003 Standard
Chapter 19. Service and utility procedures

IBM Extension

XL Fortran provides utility services that are available to the Fortran programmer. This section describes the rules for the general service and utility procedures, then provides an alphabetical reference to these procedures.

General service and utility procedures

The general service and utility procedures belong to the xlfutility module. 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 XLUTILITY module, 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 xlfutility and xlfutility_extname modules:

<table>
<thead>
<tr>
<th>File names</th>
<th>File type</th>
<th>Locations</th>
</tr>
</thead>
<tbody>
<tr>
<td>xlfutility.f</td>
<td>source file</td>
<td>/opt/ibmcmp/xlf/10.1/samples/modules</td>
</tr>
<tr>
<td>xlfutility_extname.f</td>
<td>source file</td>
<td>/opt/ibmcmp/xlf/10.1/samples/modules</td>
</tr>
<tr>
<td>xlfutility.mod</td>
<td>module symbol file</td>
<td>/opt/ibmcmp/xlf/10.1/include</td>
</tr>
</tbody>
</table>

You can use the precompiled module by adding a USE statement to your source file (see “USE” on page 414 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 section).

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 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:

<table>
<thead>
<tr>
<th>XLF-Provided Function Name</th>
<th>Common Block or External Procedure Name You Cannot Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>mclock</td>
<td>times</td>
</tr>
<tr>
<td>rand</td>
<td>irand</td>
</tr>
</tbody>
</table>

## List of service and utility procedures

This section lists the service and utility procedures available in the XLFUTILITY module.

Any application that uses the interfaces for the procedures ctime_, gmtime_, ltime_, or time_ uses the symbolic constant TIME_SIZE to specify the kind type parameter of certain intrinsic data types. The XLFUTILITY module defines TIME_SIZE.

TIME_SIZE is set to 4 for 32-bit and 64-bit applications.

**Note:** CHARACTER(n) means that you can specify any length for the variable.

### alarm_(time, func)

**Purpose**
The alarm_ function sends an alarm signal at the specified time to invoke the specified function.

**Class**
Function

**Argument type and attributes**
- **time** INTEGER(4), INTENT(IN)
- **func** A function that returns a result of type INTEGER(4).

**Result type and attributes**
INTEGER(4)

**Result value**
The returned value is the remaining time from the last alarm.
bic_(X1, X2)

**Purpose**
The bic subroutine sets bit X1 of X2 to 0. For greater portability, it is recommended that you use the IBCLR standard intrinsic procedure instead of this procedure.

**Class**
Subroutine

**Argument type and attributes**
- **X1** INTEGER(4), INTENT(IN)
  The range of X1 must be within 0 to 31, inclusive.
- **X2** INTEGER(4), INTENT(INOUT)

bis_(X1, X2)

**Purpose**
The bis subroutine sets bit X1 of X2 to 1. For greater portability, it is recommended that you use the IBSET standard intrinsic procedure instead of this procedure.

**Class**
Subroutine

**Argument type and attributes**
- **X1** INTEGER(4), INTENT(IN)
  The range of X1 must be within 0 to 31, inclusive.
- **X2** INTEGER(4), INTENT(INOUT)

bit_(X1, X2)

**Purpose**
The bit function returns the value .TRUE. if bit X1 of X2 equals 1. Otherwise, bit returns the value .FALSE. For greater portability, it is recommended that you use the BTEST standard intrinsic procedure instead of this procedure.

**Class**
Function

**Argument type and attributes**
- **X1** INTEGER(4), INTENT(IN)
  The range of X1 must be within 0 to 31, inclusive.
- **X2** INTEGER(4), INTENT(IN)
### Result type and attributes

LOGICAL(4)

### Result value

This function returns .TRUE. if bit \( X_1 \) of \( X_2 \) equals 1. Otherwise this function returns .FALSE.

---

#### clock()

**Purpose**

The `clock_` function returns the time in hh:mm:ss format. This function is different from the operating system clock function.

**Class**

Function

**Result type and attributes**

CHARACTER(8)

**Result value**

The time in hh:mm:ss format

---

#### ctime_(STR, TIME)

**Purpose**

The `ctime_` subroutine converts the system time \( TIME \) to a 26-character ASCII string and outputs the result into the first argument.

**Class**

Subroutine

**Argument type and attributes**

- **STR** CHARACTER(26), INTENT(OUT)
- **TIME** INTEGER(KIND=TIME_SIZE), INTENT(IN)

---

#### date()

**Purpose**

The `date` function returns the current date in mm/dd/yy format.

**Class**

Function

**Result type and attributes**

CHARACTER(8)
**Result value**
The current date in mm/dd/yy format

---

**dtim**e_(dtim**e**_**struct**)  

**Purpose**
The `dtim**e**_` function sets the time accounting information for the user time and system time in DTIME_STRUCT. The resolution for all timing is 1/100 of a second. The output appears in units of seconds.

**Class**
Function

**Argument type and attributes**

```plaintext
dtim**e**_**struct**
  TYPE TB_TYPE
  SEQUENCE
    REAL(4) USRTIME
    REAL(4) SYSTIME
  END TYPE
  TYPE (TB_TYPE) DTIME_STRUCT
```

**Result type and attributes**

REAL(4)

**Result value**
The returned value is the sum of the user time and the system time since the last call to `dtim**e**_`.

---

**etime**_(etime**struct**)  

**Purpose**
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 resolution for all timing is 1/100 of a second. The output appears in units of seconds.

**Class**
Function

**Argument type and attributes**

```plaintext
etime**struct**
  TYPE TB_TYPE
  SEQUENCE
    REAL(4) USRTIME
    REAL(4) SYSTIME
  END TYPE
  TYPE (TB_TYPE) ETIME_STRUCT
```

**Result type and attributes**

REAL(4)
**Result value**

The returned value is the sum of the user-elapsed time and the system-elapsed time.

---

**exit_(exit_status)**

**Purpose**

The `exit_` subroutine stops execution of the process with the exit status of `EXIT_STATUS`.

**Class**

Subroutine

**Argument type and attributes**

- `exit_status`  
  INTEGER(4)

---

**fdate_(str)**

**Purpose**

The `fdate_` subroutine returns the date and time in a 26-character ASCII string. The ASCII string is returned in argument STR.

**Class**

Subroutine

**Argument type and attributes**

- `str`  
  CHARACTER(26)

---

**fiosetup_(unit, command, argument)**

**Purpose**

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.

**Class**

Function

**Argument type and attributes**

- `unit`  
  A logical unit that is currently connected to a file  
  INTEGER(4).

- `command`  
  INTEGER(4).
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
INTEGER(4).

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.

Result type and attributes
INTEGER(4).

Result value
On successful completion, this function returns 0. Otherwise, this function returns one of the following errors:

IO_ERR_NO_RTE (1000)
The run-time environment is not running.

IO_ERR_BAD_UNIT (1001)
The specified UNIT is not connected.

IO_ERR_BAD_CMD (1002)
Invalid command.

IO_ERR_BAD_ARG (1003)
Invalid argument.

flush_(lunit)

Purpose
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 $0 \leq \text{LUNIT} \leq 2^{31}-1$.

For greater portability, use the FLUSH statement instead of this procedure.

Class
Subroutine

Argument type and attributes
lunit INTEGER(4), INTENT(IN)
The `ftell_` function returns the offset of the current byte relative to the beginning of the file associated with the specified logical unit UNIT.

The offset returned by the `ftell_` function is the result of previously completed I/O operations. No references to `ftell_` 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.

**Class**

Function

**Argument type and attributes**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type and Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>lunit</td>
<td>INTEGER(4), INTENT(IN)</td>
</tr>
</tbody>
</table>

**Result type and attributes**

INTEGER(4)

**Result value**

The offset returned by the `ftell_` 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.

If the unit is not connected, the `ftell_` function returns -1.

The `ftell64_` function allows you to query files larger than 2 gigabytes in large file enabled file systems.

**Purpose**

The `ftell64_` function returns the offset of the current byte relative to the beginning of the file associated with the specified logical unit UNIT. The `ftell64` function allows you to query files larger than 2 gigabytes in large file enabled file systems.

The offset returned by the `ftell_` function is the result of previously completed I/O operations. No references to `ftell_` 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.

**Class**

Function

**Argument type and attributes**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type and Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>lunit</td>
<td>INTEGER(4), INTENT(IN)</td>
</tr>
</tbody>
</table>

**Result type and attributes**

The offset returned by the `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.

`ftell64_` returns INTEGER(8).

**Result value**
If the unit is not connected, the `ftell64_` function returns -1.

---

**getarg(i1,c1)**

**Purpose**
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.

For greater portability, use the `GET_COMMAND_ARGUMENT` intrinsic instead of this procedure.

**Class**
Subroutine

**Argument type and attributes**
- `i1` INTEGER(4), INTENT(IN)
- `c1` CHARACTER(1024), INTENT(OUT)

`X` is the maximum number of characters `c1` can hold.

---

**getcwd_(name)**

**Purpose**
The `getcwd_` function retrieves the pathname `NAME` of the current working directory where the maximum length is 1024 characters.

**Class**
Function

**Argument type and attributes**
- `name` A character string of maximum length 1024

**Result type and attributes**
INTEGER(4)

**Result value**
On successful completion, this function returns 0. Otherwise, it returns a system error code (`errno`).
**getfd(lunit)**

**Purpose**
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 *XL Fortran Optimization and Programming Guide*.

**Class**
Function

**Argument type and attributes**

- **lunit** INTEGER(4), INTENT(IN)

**Result type and attributes**

- INTEGER(4)

**Result value**
This function returns the underlying file descriptor of the given logical unit, or -1 if the unit is not connected.

---

**getgid_()**

**Purpose**
The *getgid_* function returns the group id of a process, where GROUP_ID is the requested real group id of the calling process.

**Class**
Function

**Result type and attributes**

- INTEGER(4)

**Result value**
The group id of a process

---

**getlog_(name)**

**Purpose**
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.

**Class**
Subroutine
Argument type and attributes

name CHARACTER(8), INTENT(OUT)

getpid()

Purpose
The getpid_ function returns the process id of the current process.

Class
Function

Result type and attributes
INTEGER(4)

Result value
The process id of the current process

getuid()

Purpose
The getuid_ function returns the real user id of the current process.

Class
Function

Result type and attributes
INTEGER(4)

Result value
The real user id of the current process

global_timef()

Purpose
The global_timef function returns the elapsed time since the first call to global_timef was first executed among all running threads. For thread-specific timing results, see the timef_delta function.

Class
Function

Result type and attributes
REAL(8)


Result value

This function returns in milliseconds, the global timing results from all running threads. The first call to `global_timef` returns 0.0. The accuracy of an XL Fortran timing function is operating system dependent.


gmtime(stime, tarray)

Purpose

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)

Class

Subroutine

Argument type and attributes

- `stime`  INTEGER(KIND=TIME_SIZE), INTENT(IN)
- `tarray` INTEGER(4), INTENT(OUT) :: tarray(9)


defhostnm_(name)

Purpose

The `hostnm_` function retrieves the machine’s host name `NAME`. `NAME` has a maximum length of 63 characters.

For greater portability, use the `GET_ENVIRONMENT_VARIABLE` intrinsic instead of this procedure.

Class

Function

Argument type and attributes

- `name`  CHARACTER(X), INTENT(OUT)

  X can be in the range of 1 to 63.

Result type and attributes

INTEGER(4).

Result value

The returned value is 0 if the host name is found, and an error number otherwise.
iargc()

**Purpose**

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.

For greater portability, use the `COMMAND_ARGUMENT_COUNT` intrinsic instead of this procedure.

**Class**

Function

**Result type and attributes**

INTEGER(4)

**Result value**

The number of arguments

idate_(idate_struct)

**Purpose**

The `idate_` subroutine returns the current date in a numerical format containing the day, month and year.

**Class**

Subroutine

**Argument type and attributes**

```plaintext
idate_struct
    TYPE IDATE_TYPE
    SEQUENCE
        INTEGER(4) IDAY
        INTEGER(4) IMONTH
        INTEGER(4) IYEAR
    END TYPE
    TYPE (IDATE_TYPE) IDATE_STRUCT
```

ierrno()

**Purpose**

The `ierrno_` function returns the error number of the last detected system error.

**Class**

Function

**Result type and attributes**

INTEGER(4)
Result value

The error number of the last detected system error

irand()

Purpose

The irand function generates a positive integer greater than 0 and less than or equal to 32768. The intrinsic subroutine "SRAND(SEED)" on page 594 is used to provide the seed value for the random number generator.

Class

Function

Result type and attributes

INTEGER(4)

Result value

A pseudo-random positive integer greater than 0 and less than or equal to 32768

irtc()

Purpose

The irtc function returns the number of nanoseconds since the initial value of the machine's real-time clock.

Class

Function

Result type and attributes

INTEGER(8)

Result value

The number of nanoseconds since the initial value of the machine's real-time clock.

itime_(itime_struct)

Purpose

The itime_ subroutine returns the current time in a numerical form containing seconds, minutes, and hours in ITIME_STRUCT.

Class

Subroutine

Argument type and attributes

itime_struct

TYPE IAR
SEQUENCE
INTEGER(4) IHR
jdate()

Purpose
The jdate function returns the current Julian date in yyddd format.

Class
Function

Result type and attributes
CHARACTER(8)

Result value
The current Julian date in yyddd format

lenchr_(str)

Purpose
The lenchr_ function returns the length of the given character string.

Class
Function

Argument type and attributes
str CHARACTER(*), INTENT(IN)

Result type and attributes
INTEGER(4)

Result value
The length of the character string

lnblnk_(str)

Purpose
The lnblnk_ function returns the index of the last non-blank character in the string STR. If the string contains no non-blank characters, 0 is returned.

Class
Function

Argument type and attributes
str CHARACTER(*), INTENT(IN)
**Result type and attributes**

INTEGER(4)

**Result value**

The index of the last non-blank character in the string, or 0 if there are no non-blank characters.

---

**ltime_(stime, tarray)**

**Purpose**

The *ltime_* subroutine dissect 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)

**Class**

Subroutine

**Argument type and attributes**

- **stime**  INTEGER(KIND=TIME_SIZE), INTENT(IN)
- **tarray** INTEGER(4), INTENT(OUT):: tarry(9)

---

**mclock()**

**Purpose**

The *mclock* function returns time accounting information about the current process and its child processes. The accuracy of an XL Fortran timing function is operating system dependent.

**Class**

Function

**Result type and attributes**

INTEGER(4)

**Result value**

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.
qsort_(array, len, isize, compar)

**Purpose**

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.

**Class**

Subroutine

**Argument type and attributes**

- **array** The array to be sorted. It can be of any type.
- **len** The number of elements in the array. The argument is of type INTEGER(4).
- **isize** The size of a single element of the array. The argument is of type INTEGER(4).
- **compar** A user-defined comparison function used to sort the array.

**Examples**

```fortran
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
```

qsort_down(array, len, isize)

**Purpose**

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.

**Class**

Subroutine
Argument type and attributes

array  The array to be sorted. It can be of any type.
len    The number of elements in the array. The argument is of type INTEGER(4).
isize  The size of a single element of the array. The argument is of type INTEGER(4).

Result type and attributes

Result value

Examples

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(array, len, isize)

Purpose

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.

Class

Subroutine

Argument type and attributes

array  The array to be sorted. It can be of any type.
len    The number of elements in the array. The argument is of type INTEGER(4).
isize  The size of a single element of the array. The argument is of type INTEGER(4).

Examples

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
rtc()

**Purpose**
The rtc function returns the number of seconds since the initial value of the machine’s real-time clock.

**Class**
Function

**Result type and attributes**
REAL(8)

**Result value**
The number of seconds since the initial value of the machine’s real-time clock.

setrteopts(c1)

**Purpose**
The setrteopts subroutine changes the setting of one or more of the run-time options during the execution of a program. See Setting run-time options in the XL Fortran Compiler Reference for details about the run-time options.

**Class**
Subroutine

**Argument type and attributes**

<table>
<thead>
<tr>
<th>c1</th>
<th>CHARACTER(X), INTENT(IN)</th>
</tr>
</thead>
</table>

X is the length of the run-time option to be set.

sleep_(sec)

**Purpose**
The sleep_ subroutine suspends the execution of the current process for SEC seconds.

**Class**
Subroutine

**Argument type and attributes**

<table>
<thead>
<tr>
<th>sec</th>
<th>INTEGER(4), INTENT(IN)</th>
</tr>
</thead>
</table>

time_()

**Purpose**
The time_ function returns the current time (GMT), in seconds.
Class
Function

Result type and attributes
INTEGER(KIND=TIME_SIZE).

Result value
The current time (GMT), in seconds

_timef()

Purpose
The _timef function returns the elapsed time in milliseconds since the first call to _timef. The accuracy of an XL Fortran timing function is operating system dependent.

Class
Function

Result type and attributes
REAL(8)

Result value
The elapsed time in milliseconds since the first call to _timef. The first call to _timef returns 0.0d0.

_timef_delta(t)

Purpose
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. The accuracy of an XL Fortran timing function is operating system dependent.

Class
Function

Argument type and attributes
t REAL(8)

Result type and attributes
REAL(8)

Result value
Time elapsed in milliseconds
umask_(cmask)

**Purpose**
The `umask_` function sets the file mode creation mask to CMASK.

**Class**
Function

**Argument type and attributes**
cmask INTEGER(4), INTENT(IN)

**Result type and attributes**
INTEGER(4)

**Result value**
The returned value is the previous value of the file mode creation mask.

usleep_(msec)

**Purpose**
The `usleep_` function suspends the execution of the current process for an interval of MSEC microseconds. The accuracy of an XL Fortran timing function is operating system dependent.

**Class**
Function

**Argument type and attributes**
msec INTEGER(4), INTENT(IN)

**Result type and attributes**
INTEGER(4)

**Result value**
The returned value is 0 if the function is successful, or an error number otherwise.

xl__trbk()

**Purpose**
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.

**Class**
Subroutine

End of IBM Extension
Appendix A. Compatibility across standards

This information is provided for the benefit of users of earlier language standards, such as FORTRAN 77, who are unfamiliar with more current language standards like Fortran 95, Fortran 90, or Fortran 2003, or with XL Fortran.

Except as noted here, the Fortran 90, Fortran 95, and Fortran 2003 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 1 below regarding intrinsic procedures. Any standard-conforming FORTRAN 77 program remains standard-conforming under the Fortran 95 or Fortran 2003 standard, as long as none of the deleted features are used in the program, except as noted under item 3 below regarding intrinsic procedures. The Fortran 90, Fortran 95, and Fortran 2003 standards restrict 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, Fortran 95, or Fortran 2003 standard, yet remain a standard-conforming program. The following FORTRAN 77 features have different interpretations in Fortran 90, Fortran 95, and Fortran 2003:

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, Fortran 95, and Fortran 2003 do not permit this processor-dependent 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, Fortran 95, and Fortran 2003 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, Fortran 95, and Fortran 2003 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 -qxl77 compiler option is specified.

4. The Fortran 90, Fortran 95, and Fortran 2003 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, Fortran 95, or Fortran 2003 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 and Fortran 2003, 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 and Fortran 2003 allow 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 Fortran 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

```fortran
CALL SUB(A,B,C,*10,*20,*30)
```

! Fortran 90

```fortran
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 34. Equivalent characters in the ASCII and EBCDIC character sets

<table>
<thead>
<tr>
<th>Decimal Value</th>
<th>Hex Value</th>
<th>Control Character</th>
<th>ASCII Symbol</th>
<th>Meaning</th>
<th>EBCDIC Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>Ctrl-@</td>
<td>NUL</td>
<td>null</td>
<td>NUL</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>Ctrl-A</td>
<td>SOH</td>
<td>start of heading</td>
<td>SOH</td>
<td>start of heading</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>Ctrl-B</td>
<td>STX</td>
<td>start of text</td>
<td>STX</td>
<td>start of text</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>Ctrl-C</td>
<td>ETX</td>
<td>end of text</td>
<td>ETX</td>
<td>end of text</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>Ctrl-D</td>
<td>EOT</td>
<td>end of transmission</td>
<td>SEL</td>
<td>select</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>Ctrl-E</td>
<td>ENQ</td>
<td>enquiry</td>
<td>HT</td>
<td>horizontal tab</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>Ctrl-F</td>
<td>ACK</td>
<td>acknowledge</td>
<td>RNL</td>
<td>required new-line</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>Ctrl-G</td>
<td>BEL</td>
<td>bell</td>
<td>DEL</td>
<td>delete</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>Ctrl-H</td>
<td>BS</td>
<td>backspace</td>
<td>GE</td>
<td>graphic escape</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>Ctrl-I</td>
<td>HT</td>
<td>horizontal tab</td>
<td>SPS</td>
<td>superscript</td>
</tr>
<tr>
<td>10</td>
<td>0A</td>
<td>Ctrl-J</td>
<td>LF</td>
<td>line feed</td>
<td>RPT</td>
<td>repeat</td>
</tr>
<tr>
<td>11</td>
<td>0B</td>
<td>Ctrl-K</td>
<td>VT</td>
<td>vertical tab</td>
<td>VT</td>
<td>vertical tab</td>
</tr>
<tr>
<td>12</td>
<td>0C</td>
<td>Ctrl-L</td>
<td>FF</td>
<td>form feed</td>
<td>FF</td>
<td>form feed</td>
</tr>
<tr>
<td>13</td>
<td>0D</td>
<td>Ctrl-M</td>
<td>CR</td>
<td>carriage return</td>
<td>CR</td>
<td>carriage return</td>
</tr>
<tr>
<td>14</td>
<td>0E</td>
<td>Ctrl-N</td>
<td>SO</td>
<td>shift out</td>
<td>SO</td>
<td>shift out</td>
</tr>
<tr>
<td>15</td>
<td>0F</td>
<td>Ctrl-O</td>
<td>SI</td>
<td>shift in</td>
<td>SI</td>
<td>shift in</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>Ctrl-P</td>
<td>DLE</td>
<td>data link escape</td>
<td>DLE</td>
<td>data link escape</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>Ctrl-Q</td>
<td>DC1</td>
<td>device control 1</td>
<td>DC1</td>
<td>device control 1</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>Ctrl-R</td>
<td>DC2</td>
<td>device control 2</td>
<td>DC2</td>
<td>device control 2</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>Ctrl-S</td>
<td>DC3</td>
<td>device control 3</td>
<td>DC3</td>
<td>device control 3</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>Ctrl-T</td>
<td>DC4</td>
<td>device control 4</td>
<td>RES/ENP</td>
<td>restore/enable presentation</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>Ctrl-U</td>
<td>NAK</td>
<td>negative acknowledgement</td>
<td>NL</td>
<td>new-line</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>Ctrl-V</td>
<td>SYN</td>
<td>synchronous idle</td>
<td>BS</td>
<td>backspace</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>Ctrl-W</td>
<td>ETB</td>
<td>end of transmission block</td>
<td>POC</td>
<td>program-operator communications</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>Ctrl-X</td>
<td>CAN</td>
<td>cancel</td>
<td>CAN</td>
<td>cancel</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>Ctrl-Y</td>
<td>EM</td>
<td>end of medium</td>
<td>EM</td>
<td>end of medium</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>Ctrl-Z</td>
<td>SUB</td>
<td>substitute</td>
<td>UBS</td>
<td>unit backspace</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>Ctrl-[</td>
<td>ESC</td>
<td>escape</td>
<td>CU1</td>
<td>customer use 1</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>Ctrl-\</td>
<td>FS</td>
<td>file separator</td>
<td>IFS</td>
<td>interchange file separator</td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>Ctrl-]</td>
<td>GS</td>
<td>group separator</td>
<td>IGS</td>
<td>interchange group separator</td>
</tr>
</tbody>
</table>

© Copyright IBM Corp. 1990, 2005
<table>
<thead>
<tr>
<th>Decimal Value</th>
<th>Hex Value</th>
<th>Control Character</th>
<th>ASCII Symbol</th>
<th>Meaning</th>
<th>EBCDIC Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>1E</td>
<td>Ctrl-^A</td>
<td>RS</td>
<td>record separator</td>
<td>IRS</td>
<td>interchange record separator</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>Ctrl-_</td>
<td>US</td>
<td>unit separator</td>
<td>IUS/ITB</td>
<td>interchange unit separator / intermediate transmission block</td>
</tr>
<tr>
<td>32</td>
<td>20</td>
<td>SP</td>
<td>space</td>
<td>DS</td>
<td>digit select</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>21</td>
<td>!</td>
<td>exclamation mark</td>
<td>SOS</td>
<td>start of significance</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>22</td>
<td>&quot;</td>
<td>straight double quotation mark</td>
<td>FS</td>
<td>field separator</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>23</td>
<td>#</td>
<td>number sign</td>
<td>WUS</td>
<td>word underscore</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>24</td>
<td>$</td>
<td>dollar sign</td>
<td>BYP/INP</td>
<td>bypass/inhibit presentation</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>25</td>
<td>%</td>
<td>percent sign</td>
<td>LF</td>
<td>line feed</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>26</td>
<td>&amp;</td>
<td>ampersand</td>
<td>ETB</td>
<td>end of transmission block</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>27</td>
<td>'</td>
<td>apostrophe</td>
<td>ESC</td>
<td>escape</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>28</td>
<td>(</td>
<td>left parenthesis</td>
<td>SA</td>
<td>set attribute</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>29</td>
<td>)</td>
<td>right parenthesis</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>2A</td>
<td>*</td>
<td>asterisk</td>
<td>SM/SW</td>
<td>set model switch</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>2B</td>
<td>+</td>
<td>addition sign</td>
<td>CSP</td>
<td>control sequence prefix</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>2C</td>
<td>,</td>
<td>comma</td>
<td>MFA</td>
<td>modify field attribute</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>2D</td>
<td>-</td>
<td>subtraction sign</td>
<td>ENQ</td>
<td>enquiry</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>2E</td>
<td>.</td>
<td>period</td>
<td>ACK</td>
<td>acknowledge</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>2F</td>
<td>/</td>
<td>right slash</td>
<td>BEL</td>
<td>bell</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>30</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>31</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>32</td>
<td>2</td>
<td></td>
<td>SYN</td>
<td>synchronous idle</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>33</td>
<td>3</td>
<td></td>
<td>IR</td>
<td>index return</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>34</td>
<td>4</td>
<td></td>
<td>PP</td>
<td>presentation position</td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>35</td>
<td>5</td>
<td></td>
<td>TRN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>36</td>
<td>6</td>
<td></td>
<td>NBS</td>
<td>numeric backspace</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>37</td>
<td>7</td>
<td></td>
<td>EOT</td>
<td>end of transmission</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>38</td>
<td>8</td>
<td></td>
<td>SBS</td>
<td>subscript</td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>39</td>
<td>9</td>
<td></td>
<td>IT</td>
<td>indent tab</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>3A</td>
<td>:</td>
<td>colon</td>
<td>RFF</td>
<td>required form feed</td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>3B</td>
<td>;</td>
<td>semicolon</td>
<td>CU3</td>
<td>customer use 3</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>3C</td>
<td>&lt;</td>
<td>less than</td>
<td>DC4</td>
<td>device control 4</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>3D</td>
<td>=</td>
<td>equal</td>
<td>NAK</td>
<td>negative acknowledge</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>3E</td>
<td>&gt;</td>
<td>greater than</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>3F</td>
<td>?</td>
<td>question mark</td>
<td>SUB</td>
<td>substitute</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>40</td>
<td>@</td>
<td>at symbol</td>
<td>SP</td>
<td>space</td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>41</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>42</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>43</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>44</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>45</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>46</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 34. Equivalent characters in the ASCII and EBCDIC character sets (continued)

<table>
<thead>
<tr>
<th>Decimal Value</th>
<th>Hex Value</th>
<th>Control Character</th>
<th>ASCII Symbol</th>
<th>Meaning</th>
<th>EBCDIC Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>71</td>
<td>47</td>
<td></td>
<td>G</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>48</td>
<td></td>
<td>H</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>73</td>
<td>49</td>
<td></td>
<td>I</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>74</td>
<td>4A</td>
<td></td>
<td>J</td>
<td>c</td>
<td>cent</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>4B</td>
<td></td>
<td>K</td>
<td>.</td>
<td>period</td>
<td></td>
</tr>
<tr>
<td>76</td>
<td>4C</td>
<td></td>
<td>L</td>
<td>&lt;</td>
<td>less than</td>
<td></td>
</tr>
<tr>
<td>77</td>
<td>4D</td>
<td></td>
<td>M</td>
<td>(</td>
<td>left parenthesis</td>
<td></td>
</tr>
<tr>
<td>78</td>
<td>4E</td>
<td></td>
<td>N</td>
<td>+</td>
<td>addition sign</td>
<td></td>
</tr>
<tr>
<td>79</td>
<td>4F</td>
<td></td>
<td>O</td>
<td>!</td>
<td>logical or</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>50</td>
<td></td>
<td>P</td>
<td>&amp;</td>
<td>ampersand</td>
<td></td>
</tr>
<tr>
<td>81</td>
<td>51</td>
<td></td>
<td>Q</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>82</td>
<td>52</td>
<td></td>
<td>R</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>83</td>
<td>53</td>
<td></td>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>84</td>
<td>54</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>85</td>
<td>55</td>
<td></td>
<td>U</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>86</td>
<td>56</td>
<td></td>
<td>V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>87</td>
<td>57</td>
<td></td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>88</td>
<td>58</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>89</td>
<td>59</td>
<td></td>
<td>Y</td>
<td>!</td>
<td>exclamation mark</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>5A</td>
<td></td>
<td>Z</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>91</td>
<td>5B</td>
<td>[</td>
<td>left bracket</td>
<td>$</td>
<td>dollar sign</td>
<td></td>
</tr>
<tr>
<td>92</td>
<td>5C</td>
<td>\</td>
<td>left slash</td>
<td>*</td>
<td>asterisk</td>
<td></td>
</tr>
<tr>
<td>93</td>
<td>5D</td>
<td>]</td>
<td>right bracket</td>
<td>)</td>
<td>right parenthesis</td>
<td></td>
</tr>
<tr>
<td>94</td>
<td>5E</td>
<td>^</td>
<td>hat, circumflex</td>
<td>;</td>
<td>semicolon</td>
<td></td>
</tr>
<tr>
<td>95</td>
<td>5F</td>
<td>_</td>
<td>underscore</td>
<td>~</td>
<td>logical not</td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>60</td>
<td>`</td>
<td>grave</td>
<td>-</td>
<td>subtraction sign</td>
<td></td>
</tr>
<tr>
<td>97</td>
<td>61</td>
<td>a</td>
<td>/</td>
<td>right slash</td>
<td></td>
<td></td>
</tr>
<tr>
<td>98</td>
<td>62</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>63</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>64</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>65</td>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>102</td>
<td>66</td>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>103</td>
<td>67</td>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>68</td>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>69</td>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>106</td>
<td>6A</td>
<td>j</td>
<td>!</td>
<td>split vertical bar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>107</td>
<td>6B</td>
<td>k</td>
<td>,</td>
<td>comma</td>
<td></td>
<td></td>
</tr>
<tr>
<td>108</td>
<td>6C</td>
<td>l</td>
<td>%</td>
<td>percent sign</td>
<td></td>
<td></td>
</tr>
<tr>
<td>109</td>
<td>6D</td>
<td>m</td>
<td>_</td>
<td>underscore</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>6E</td>
<td>n</td>
<td>&gt;</td>
<td>greater than</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>6F</td>
<td>o</td>
<td>?</td>
<td>question mark</td>
<td></td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>70</td>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>113</td>
<td>71</td>
<td>q</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>114</td>
<td>72</td>
<td>r</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decimal Value</td>
<td>Hex Value</td>
<td>Control Character</td>
<td>ASCII Symbol</td>
<td>Meaning</td>
<td>EBCDIC Symbol</td>
<td>Meaning</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------</td>
<td>-------------------</td>
<td>--------------</td>
<td>---------</td>
<td>---------------</td>
<td>---------------</td>
</tr>
<tr>
<td>115</td>
<td>73</td>
<td></td>
<td>s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>116</td>
<td>74</td>
<td></td>
<td>t</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>117</td>
<td>75</td>
<td></td>
<td>u</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>118</td>
<td>76</td>
<td></td>
<td>v</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>119</td>
<td>77</td>
<td></td>
<td>w</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>78</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>121</td>
<td>79</td>
<td></td>
<td>y</td>
<td></td>
<td>`</td>
<td>grave</td>
</tr>
<tr>
<td>122</td>
<td>7A</td>
<td></td>
<td>z</td>
<td></td>
<td>:</td>
<td>colon</td>
</tr>
<tr>
<td>123</td>
<td>7B</td>
<td></td>
<td></td>
<td></td>
<td>#</td>
<td>numbersign</td>
</tr>
<tr>
<td>124</td>
<td>7C</td>
<td></td>
<td></td>
<td></td>
<td>@</td>
<td>at symbol</td>
</tr>
<tr>
<td>125</td>
<td>7D</td>
<td></td>
<td></td>
<td></td>
<td>’</td>
<td>apostrophe</td>
</tr>
<tr>
<td>126</td>
<td>7E</td>
<td></td>
<td></td>
<td></td>
<td>=</td>
<td>equal</td>
</tr>
<tr>
<td>127</td>
<td>7F</td>
<td>DEL</td>
<td>delete</td>
<td></td>
<td>”</td>
<td>straight double quotation mark</td>
</tr>
<tr>
<td>128</td>
<td>80</td>
<td></td>
<td>a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>129</td>
<td>81</td>
<td></td>
<td>b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>130</td>
<td>82</td>
<td></td>
<td>c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>131</td>
<td>83</td>
<td></td>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>132</td>
<td>84</td>
<td></td>
<td>e</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>133</td>
<td>85</td>
<td></td>
<td>f</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>134</td>
<td>86</td>
<td></td>
<td>g</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>135</td>
<td>87</td>
<td></td>
<td>h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>136</td>
<td>88</td>
<td></td>
<td>i</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>137</td>
<td>89</td>
<td></td>
<td>j</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>138</td>
<td>8A</td>
<td></td>
<td>k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>139</td>
<td>8B</td>
<td></td>
<td>l</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>140</td>
<td>8C</td>
<td></td>
<td>m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>141</td>
<td>8D</td>
<td></td>
<td>n</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>142</td>
<td>8E</td>
<td></td>
<td>o</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>143</td>
<td>8F</td>
<td></td>
<td>p</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>144</td>
<td>90</td>
<td></td>
<td>q</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>145</td>
<td>91</td>
<td></td>
<td>r</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>146</td>
<td>92</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>147</td>
<td>93</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>148</td>
<td>94</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>149</td>
<td>95</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>150</td>
<td>96</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>151</td>
<td>97</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>152</td>
<td>98</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>153</td>
<td>99</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>154</td>
<td>9A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>155</td>
<td>9B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>156</td>
<td>9C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>157</td>
<td>9D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 34. Equivalent characters in the ASCII and EBCDIC character sets  (continued)

<table>
<thead>
<tr>
<th>Decimal Value</th>
<th>Hex Value</th>
<th>Control Character</th>
<th>ASCII Symbol</th>
<th>Meaning</th>
<th>EBCDIC Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>158</td>
<td>9E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>159</td>
<td>9F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>160</td>
<td>A0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>161</td>
<td>A1</td>
<td>~</td>
<td>similar, tilde</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>162</td>
<td>A2</td>
<td>s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>163</td>
<td>A3</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>164</td>
<td>A4</td>
<td>u</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>165</td>
<td>A5</td>
<td>v</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>166</td>
<td>A6</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>167</td>
<td>A7</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>168</td>
<td>A8</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>169</td>
<td>A9</td>
<td>z</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>170</td>
<td>AA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>171</td>
<td>AB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>172</td>
<td>AC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>173</td>
<td>AD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>174</td>
<td>AE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>175</td>
<td>AF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>176</td>
<td>B0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>177</td>
<td>B1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>178</td>
<td>B2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>179</td>
<td>B3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>180</td>
<td>B4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>181</td>
<td>B5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>182</td>
<td>B6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>183</td>
<td>B7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>184</td>
<td>B8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>185</td>
<td>B9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>186</td>
<td>BA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>187</td>
<td>BB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>188</td>
<td>BC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>189</td>
<td>BD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>190</td>
<td>BE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>191</td>
<td>BF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>192</td>
<td>C0</td>
<td>{</td>
<td>left brace</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>193</td>
<td>C1</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>194</td>
<td>C2</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>195</td>
<td>C3</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>196</td>
<td>C4</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>197</td>
<td>C5</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>198</td>
<td>C6</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>199</td>
<td>C7</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>C8</td>
<td>H</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>201</td>
<td>C9</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 34. Equivalent characters in the ASCII and EBCDIC character sets (continued)

<table>
<thead>
<tr>
<th>Decimal Value</th>
<th>Hex Value</th>
<th>Control Character</th>
<th>ASCII Symbol</th>
<th>Meaning</th>
<th>EBCDIC Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>202</td>
<td>CA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>203</td>
<td>CB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>204</td>
<td>CC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>205</td>
<td>CD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>206</td>
<td>CE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>207</td>
<td>CF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>208</td>
<td>D0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>209</td>
<td>D1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>210</td>
<td>D2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>211</td>
<td>D3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>212</td>
<td>D4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>213</td>
<td>D5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>214</td>
<td>D6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>215</td>
<td>D7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>216</td>
<td>D8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>217</td>
<td>D9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>218</td>
<td>DA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>219</td>
<td>DB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>220</td>
<td>DC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>221</td>
<td>DD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>222</td>
<td>DE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>223</td>
<td>DF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>224</td>
<td>E0</td>
<td></td>
<td>\</td>
<td>left slash</td>
<td></td>
<td></td>
</tr>
<tr>
<td>225</td>
<td>E1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>226</td>
<td>E2</td>
<td></td>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>227</td>
<td>E3</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>228</td>
<td>E4</td>
<td></td>
<td>U</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>229</td>
<td>E5</td>
<td></td>
<td>V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>230</td>
<td>E6</td>
<td></td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>231</td>
<td>E7</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>232</td>
<td>E8</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>233</td>
<td>E9</td>
<td></td>
<td>Z</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>234</td>
<td>EA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>235</td>
<td>EB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>236</td>
<td>EC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>237</td>
<td>ED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>238</td>
<td>EE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>239</td>
<td>EF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>240</td>
<td>F0</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>241</td>
<td>F1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>242</td>
<td>F2</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>243</td>
<td>F3</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>244</td>
<td>F4</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>245</td>
<td>F5</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 34. Equivalent characters in the ASCII and EBCDIC character sets (continued)

<table>
<thead>
<tr>
<th>Decimal Value</th>
<th>Hex Value</th>
<th>Control Character</th>
<th>ASCII Symbol</th>
<th>Meaning</th>
<th>EBCDIC Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>246</td>
<td>F6</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>247</td>
<td>F7</td>
<td></td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>248</td>
<td>F8</td>
<td></td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>249</td>
<td>F9</td>
<td></td>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>250</td>
<td>FA</td>
<td></td>
<td>1</td>
<td></td>
<td>vertical line</td>
<td></td>
</tr>
<tr>
<td>251</td>
<td>FB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>252</td>
<td>FC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>253</td>
<td>FD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>254</td>
<td>FE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>255</td>
<td>FF</td>
<td></td>
<td>EO</td>
<td>eight ones</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 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
8200 Warden Avenue
Markham, Ontario, Canada
L6G 1C7

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.

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.
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:

IBM  IBM(logo)  POWER4
POWER5  PowerPC  pSeries

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.
Glossary

This glossary defines terms that are commonly used in this document. It includes definitions developed by the American National Standards Institute (ANSI) and entries from the IBM Dictionary of Computing.

A

active processor. See online processor.

actual argument. An expression, variable, procedure, or alternate return specifier that is specified in a procedure reference.

alias. A single piece of storage that can be accessed through more than a single name. Each name is an alias for that storage.

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 $ and _) 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. See ASCII.

argument. An expression that is passed to a function or subroutine. See also actual argument, dummy argument.

argument association. The relationship between an actual argument and a dummy argument during the invocation of a procedure.

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 and one or more subscripts. See also subscript.

array name. The name of an ordered set of data items.

array section. A subobject that is an array and is not a structure component.

ASCII. The standard code, using a coded character set consisting of 7-bit coded characters (8-bits including parity check), that is used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters. See also Unicode.

asynchronous. Pertaining to events that are not synchronized in time or do not occur in regular or predictable time intervals.

assignment statement. An executable statement that defines or redefines a variable based on the result of expression evaluation.

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 and DO loops 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.

bss storage. Uninitialized static storage.

busy-wait. The state in which a thread keeps executing in a tight loop looking for more work once it has completed all of its work and there is no new work to do.

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 the characters are ordered for the purpose of sorting, merging, comparing, and processing indexed data sequentially.

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.

compile. To translate a source program into an executable program (an object program).

compiler comment directive. A line in source code that is not a Fortran statement but is recognized and acted on by the compiler.

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.

conform. To adhere to a prevailing standard. An executable program conforms to the Fortran 95 Standard if it uses only those forms and relationships described therein and if the executable program has an interpretation according to the Fortran 95 Standard. A program unit conforms to the Fortran 95 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. The four classes of constants specify numbers (arithmetic), truth values (logical), character data (character), and typeless data (hexadecimal, octal, and binary). See also variable.

construct. A sequence of statements starting with a SELECT CASE, DO, IF, or WHERE statement, for example, and ending with the corresponding terminal statement.

continuation line. A line that 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 striping. Spreading data across multiple storage devices so that I/O operations can be performed in parallel for better performance. Also known as disk striping.

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. See also intrinsic.

default initialization. The initialization of an object with a value specified as part of a derived type definition.

definable variable. A variable whose 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.

denormalized number. An IEEE number with a very small absolute value and lowered precision. A denormalized number is represented by a zero exponent and a non-zero fraction.

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.

disk striping. See data striping.

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 loop. 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 dimensioning. The process of re-evaluating the bounds of an array each time the array is referenced.

dynamic extent. For a directive, the lexical extent of the directive and all subprograms called from within the lexical extent.

E

edit descriptor. An abbreviated keyword that controls the formatting of integer, real, or complex data.

elemental. Pertaining 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 blank. A blank that is surrounded by any other characters.

entity. A general term for any of 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.

environment variable. A variable that describes the operating environment of the process.

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 perform a calculation, test conditions, or alter normal sequential execution.

explicit initialization. The initialization of an object with a value in a data statement initial value list, block data program unit, type declaration statement, or array constructor.

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, a constant, or a 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 file. A sequence of records on an input/output device. See also internal file.

external name. The name of a common block, subroutine, or other global procedure, which the linker uses to resolve references from one compilation unit to another.

external procedure. A procedure that is defined by an external subprogram or by a means other than Fortran.

field. An area in a record used to contain a particular category of data.

file. A sequence of records. See also external file, internal file.

file index. See i-node.

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 and unformatted record.

function. A procedure that returns the value of a single variable or an object and usually has a single exit. See also intrinsic procedure, subprogram.

G

generic identifier. A lexical token that appears in an INTERFACE statement and is associated with all the procedures in an interface block.

H

hard limit. A system resource limit that can only be raised or lowered by using root authority, or cannot be altered because it is inherent in the system or operating environments’s implementation. See also soft limit.

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.

high order transformations. A type of optimization that restructures loops and array language.

Hollerith constant. A string of any characters capable of representation by XL Fortran and preceded with $\text{H}$, 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.

IPA. Interprocedural analysis, a type of optimization that allows optimizations to be performed across procedure boundaries and across calls to procedures in separate source files.

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.

infinity. An IEEE number (positive or negative) created by overflow or division by zero. Infinity is represented by an exponent where all the bits are 1’s, and a zero fraction.

i-node. The internal structure that describes the individual files in the operating system. There is at least one i-node for each file. An i-node contains the node, type, owner, and location of a file. A table of i-nodes is stored near the beginning of a file system. Also known as file index.

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 its corresponding END INTERFACE statement.

interface body.  A sequence of statements in an interface block from a FUNCTION or SUBROUTINE statement to its corresponding END statement.

interference.  A situation in which two iterations within a DO loop have dependencies upon one another.

internal file.  A sequence of records in internal storage. See also external file.

interprocedural analysis.  See IPA.

intrinsic.  Pertaining to types, operations, assignment statements, and procedures that are defined by Fortran language standards and can be used in any scoping unit without further definition or specification.

intrinsic module.  A module that is provided by the compiler and is available to any program.

intrinsic procedure.  A procedure that is provided by the compiler and is available to any program.

K

keyword.  (1) A statement keyword is a word that is part of the syntax of a statement (or directive) and may be used to identify the statement. (2) An argument keyword specifies the name of a dummy argument.

kind type parameter.  A parameter whose values label the available kinds of an intrinsic type.

L

lexical extent.  All of the code that appears directly within a directive construct.

lexical token.  A sequence of characters with an indivisible interpretation.

link-edit.  To create a loadable computer program by means of a linker.

linker.  A program that resolves cross-references between separately compiled or assembled object modules and then assigns final addresses to create a single relocatable load module. If a single object module is linked, the linker simply makes it relocatable.

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.  A lexical token that directly represents a scalar value of intrinsic type.

load balancing.  An optimization strategy that aims at evenly distributing the work load among processors.

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.  The default name given to a main program by the compiler if the main program was not named by the programmer.

main program.  The first program unit to receive control when a program is run. See also 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.  A primitive object that provides mutual exclusion between threads. A mutex is used cooperatively between threads to ensure that only one of the cooperating threads is allowed to access shared data or run certain application code at a time.

N

NaN (not-a-number).  A symbolic entity encoded in floating-point format that does not correspond to a number. See also quiet NaN, signalling NaN.

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.
**negative zero.** An IEEE representation where the exponent and fraction are both zero, but the sign bit is 1. Negative zero is treated as equal to positive zero.

**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.

**normal.** A floating point number that is not denormal, infinity, or NaN.

**not-a-number.** See NaN.

**numeric constant.** A constant that expresses an integer, real, complex, or byte number.

**numeric storage unit.** The space occupied by a nonpointer scalar object of type default integer, default real, or default logical.

**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.

**one-trip DO-loop.** A DO loop that is executed at least once, if reached, even if the iteration count is equal to 0. (This type of loop is from FORTRAN 66.)

**online processor.** In a multiprocessor machine, a processor that has been activated (brought online). The number of online processors is less than or equal to the number of physical processors actually installed in the machine. Also known as active processor.

**operator.** 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.

**paging space.** Disk storage for information that is resident in virtual memory but is not currently being accessed.

**PDF.** See profile-directed feedback.

**pointee array.** An explicit-shape or assumed-size array that is declared in an integer POINTER statement or other specification statement.

**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 a subroutine. It may be an intrinsic procedure, an external procedure, a module procedure, an internal procedure, a dummy procedure, or a statement function. A subprogram may define more than one procedure if it contains ENTRY statements.

**procedure pointer.** A procedure entity that has the EXTERNAL and POINTER attributes. It can be pointer associated with an external procedure, a module procedure, a dummy procedure or another procedure pointer.

**profile-directed feedback (PDF).** A type of optimization that uses information collected during application execution to improve performance of conditional branches and in frequently executed sections of code.

**program unit.** A main program or subprogram.

**pure.** An attribute of a procedure that indicates there are no side effects.

**Q**

**quiet NaN.** A NaN (not-a-number) value that does not signal an exception. The intent of a quiet NaN is to propagate a NaN result through subsequent computations. See also NaN, signalling NaN.
random access. An access method in which records can be read from, written to, or removed from a file in any order. See also sequential access.

rank. 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
- .LT. less than
- .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 specified 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.

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. The object that is associated with the associate name in an ASSOCIATE construct.

semantics. The relationships of characters or groups of characters to their meanings, independent of the manner of their interpretation and use. See also syntax.

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. See also random access.

signaling NaN. A NaN (not-a-number) value that signals an invalid operation exception whenever it appears as an operand. The intent of the signaling NaN is to catch program errors, such as using an uninitialized variable. See also NaN, quiet NaN.

sleep. The state in which a thread completely suspends execution until another thread signals it that there is work to do.

SMP. See symmetric multiprocessing.

soft limit. A system resource limit that is currently in effect for a process. The value of a soft limit can be raised or lowered by a process, without requiring root authority. The soft limit for a resource cannot be raised above the setting of the hard limit. See also hard limit.

spill space. The stack space reserved in each subprogram in case there are too many variables to hold in registers and the program needs temporary storage for register contents.

specification statement. A statement that provides information about the data used in the source program. The statement could also supply information to allocate data storage.

stanza. A group of lines in a file that together have a common function or define a part of the system. Stanzas are usually separated by blank lines or colons, and each stanza has a name.

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 a 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. See also main program.

subroutine. A procedure that is invoked by a CALL statement or defined assignment statement.

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). A system in which functionally-identical multiple processors are used in parallel, providing simple and efficient load-balancing.

synchronous. Pertaining to an operation that occurs regularly or predictably with regard to the occurrence of a specified event in another process.

syntax. The rules for the construction of a statement. See also semantics.

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 stream of computer instructions that is in control of a process. A multithread process begins with one stream of instructions (one thread) and may later create other instruction streams to perform tasks.

thread visible variable. A variable that can be accessed by more than one thread.

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. In a programming language, a character string, in a particular format, that has some defined significance.

trigger constant. A sequence of characters that identifies comment lines as compiler comment directives.


type declaration statement. A statement that specifies the type, length, and attributes of an object or function. Objects can be assigned initial values.

U

unformatted record. A record that is transmitted unchanged between internal and external storage.

Unicode. A universal character encoding standard that supports the interchange, processing, and display of text that is written in any of the languages of the modern world. It also supports many classical and historical texts in a number of languages. The Unicode standard has a 16-bit international character set defined by ISO 10646. See also ASCII.

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.

unsafe option. Any option that could result in undesirable results if used in the incorrect context. Other options may result in very small variations from the default result, which is usually acceptable. Typically, using an unsafe option is an assertion that your code is not subject to the conditions that make the option unsafe.

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.

X


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

_OPENMP C preprocessor macro 12
; statement separator 8, 10
: (colon) editing 219
:: (double colon) separator 250
! inline comments 6, 7
/ (slash) editing 218
// (concatenation) operator 91
’ (apostrophe) editing 199
"" (double quotation mark) editing 199
$ (dollar) editing 219
* comment lines 7
@PROCESS 453
%VAL and %REF functions 161
#LINE 447
+, -, *, /, ** arithmetic operators 89

A
A (character) editing 202
ABORT intrinsic subroutine 478
ABS
initializing expressions 85
intrinsic function 478
specific name 479
ACCESS specifier
of INQUIRE statement 330
of OPEN statement 355
access, inquiring about 330
accessibility
private 369
public 376
ACHAR intrinsic function 479
ACOS
intrinsic function 480
specific name 480
ACOSD
intrinsic function 480
specific name 481
ACTION specifier
of INQUIRE statement 330
of OPEN statement 355
actual arguments
definition 759
specification 157
specifying procedure names as arithmetic operator 89
ADJUSTL intrinsic function 481
ADJSTR intrinsic function 481
ADVANCE specifier
of READ statement 378
of WRITE statement 425
AIMAG
initializing expressions 85
intrinsic function 482
specific name 483
AINT
intrinsic function 483
specific name 483
alarm_service and utility
subprogram 722
ALGAMA specific name 538
ALIGNX PowerPC intrinsic function 483
ALL array intrinsic function 484
ALLOCATABLE attribute 238
ALLOCATE statement 240
ALLOCATED array intrinsic function 241, 485
allocation status 58
ALOG specific name 543
ALOG10 specific name 544
alphabetical character, definition of alphanumeric, definition of alternate entry point 300
alternate return point 159
specifier 158, 168
AMAX0 specific name 548
AMAX1 specific name 548
AMIN0 specific name 553
AMIN1 specific name 553
AMOD specific name 557
AND logical operator 92
AND specific name 525
ANINT intrinsic function 486
ANINT specific name 486
ANY array intrinsic function 486
apostrophe (‘) editing 199
arguments
See also actual arguments, dummy argument
definition of 759
keywords 158
specification 157
arithmetic expressions 88
operators 89
relational expressions 95
type
complex 22, 23
integer 17
real 19
arithmetic IF statement 324
arrays
adjustable 67
allocatable 69
array pointers 70
assumed-shape 67
assumed-size 70, 87
automatic 66
bounds 63
constructors 79
declarators 65
description 63
deferred-shape 68
elements 72
exents 64
pointee 67
pointer 70
arrays (continued)
rank 64
sections 73
shape 64
size 64
specification of 65
zero-sized 63
ASCII
character set 3, 747
definition of 759
ASIN
intrinsic function 487
specific name 488
ASIND
intrinsic function 488
specific name 488
ASSERT 436
ASSIGN statement 242
assigned GO TO statement 321
assignment
defined 147
intrinsic 100
masked array 105
pointer 112
procedure pointer 114
statements
described 100
statement label (ASSIGN) 242
ASSOCIATE
construct 117
statement 243
ASSOCIATED intrinsic function 241, 488
association
argument 160
common 264
description 134
equivalence 305
host 134
integer pointer 137
pointer 136
use 135
asterisk as dummy argument 159, 168
ASYNCH specifier
of INQUIRE statement 330
of OPEN statement 355
asynchronous I/O
data transfer and 183
INQUIRE statement and 331
OPEN statement and 356
READ statement and 380
WAIT statement and 422
WRITE statement and 428
ATAN
intrinsic function 489
specific name 490
ATAN2
intrinsic function 491
specific name 492
ATAN2D
intrinsic function 492
ATAN2D (continued)
  specific name 493
ATAND
  intrinsic function 490
  specific name 491
attributes
  ALLOCATABLE 238
  AUTOMATIC 244
  BIND 247
description 238
  DIMENSION 279
  EXTERNAL 308
  INTENT 341
  INTRINSIC 344
  OPTIONAL 360
  PARAMETER 362
  POINTER 364
  PRIVATE 369
  PROTECTED 374
  PUBLIC 376
  SAVE 393
  STATIC 399
  TARGET 403
  VALUE 417
  VOLATILE 419
AUTOMATIC attribute 244
automatic object 16

B
B (binary) editing 203
BACKSPACE statement 246
bic_service and utility subprogram 723
binary
  constants 47
  editing (B) 203
  operations 83
BIND attribute 247
BIND statement 247
bis_service and utility subprogram 723
bit_service and utility subprogram 723
BIT_SIZE
  intrinsic function 86, 493
  intrinsic, constant expressions and 84
  blank statement 256
  common block 263
  editing 220
  interpretation during formatting, setting 220
  null (BN) editing 220
  specifier
    of INQUIRE statement (BLANK) 330
    of OPEN statement (BLANK) 355
  zero (BZ) editing 220
block
  ELSE 124
  ELSE IF 124
  IF 124, 325
  statement 117
block data
  program unit 153
  statement (BLOCK DATA) 248
BLOCKLOOP 438
BN (blank null) editing 220
branching control 127
BTEST
  intrinsic function 493
  specific name 494
  byte named constants 100
  BYTE type declaration statement 249
  BZ (blank zero) editing 220
C
C_ASSOCIATED intrinsic procedure 683
C_F_POINTER intrinsic procedure 684
C_FUNLOC intrinsic procedure 684
C_LOC intrinsic procedure 684
CABS specific name 479
CACHE_ZERO compiler directive 465
CALL statement 252
CASE
  construct 125, 254
  statement 254
CCOS specific name 499
CDABS specific name 479
CDCOS specific name 499
CDEXP specific name 515
CDLOG specific name 543
CDSIGN specific name 588
CDSQRT specific name 594
CEILING intrinsic function 494
CEXP specific name 515
CHAR
  intrinsic function 495
  specific name 496
character
  editing
    (A) 202
    (Q), count 215
  expressions 91
  format specification 316
  multibyte 26
  operator 91
  relational expressions 95
  set 3
  string edit descriptor 199
  substrings 27
CHARACTER
  type declaration statement 256
CHARACTER_STORAGE_SIZE 687
 CHARACTER-string editing 199
CHTZ command 505
CHUNK
  definition of 760
  clock_service and utility
    subprogram 724
CLOSE statement 260
CLR_FPSCR_FLAGS subprogram 693
CMPLX
  initialising expressions 85
  intrinsic function 496
  specific name 497
CNCALL 440
CVNERR run-time option
  conversion errors and 191
  implied-DO list and 384, 430
COLLAPSE 441
collating sequence 3
  colon (;) editing 219
COMMAND_ARGUMENT_COUNT
  intrinsic function 497
comment lines
  description 6
  fixed source form format 7
  free source form input format 9
  order within a program unit 14
common
  association 264
  block 4, 262
COMMON statement 262
communication between program units
  using arguments 157
  using common blocks 262
  using modules 149
compatibility across standards 743
compiler directives
  Sec directives
  compiler options
    -I 443
    -qalias 161
    -quotdbl 478
    -qci 443
    -qctyplss
      and the CASE statement 255
      character constants and 26, 99
typeless constants and 49
    -qdim 67, 366
    -qdirective 455
    -qdirnames 8
    -qescape
      and Hollerith constants 48
      apostrophe editing and 199
      double quotation mark editing and 199
    -H editing and 211
    -qextname 721
    -qfixed 7
    -qjintlog 98, 144
    -qjintsizx
      integer default size and 17, 24
      intrinsic procedure return types and 478
    -qlogg 98
    -qmbcs 199, 211
    -qmixed 4, 443
    -qnoscpc 26
    -qnosave 59, 328
    -qnullterm 26
    -qposion 185, 355
    -qpoint 215
    -qrealsize 18, 478
    -qsecnr 171
      CALL statement and 253
      ENTRY statement and 302
      FUNCTION statement and 320
    -qsav 59, 328
    -qsigntrap 587
    -qundef 327
    -qsflag-oldtab 7
    -qsif77
      binary editing and 203, 209, 217
      hexadecimal editing and 217
      octal editing and 214
      OPEN statement and 359
      real and complex editing and 210, 211
compiler options (continued)
 -qif90  197, 585
 -qzerosize  27
 -U  721
 -qclines  12
 complex
data type  23
complex data type  22
complex editing  195
COMPLEX type declaration statement  266
component designator  34
computed GO TO statement  322
concatenation operator  91
conditional
 INCLUDE  443
vector merge intrinsic functions  503
conditional compilation  11
conformable arrays  81, 473
CONJG
 initializing expressions  85
intrinsic function  497
specific name  498
conjunction, logical  92
constants
arithmetic
 complex  22, 23
 integer  17
 real  19
binary  47
byte named  100
character  25
description  16
expressions  84
hexadecimal  46
Hollerith  48
logical  24
octal  47
type parameters, specifiers and  15
typeless  46
construct
ASSOCIATE  117
CASE  125
DO  119
DO WHILE  123
FORALL  109
IF  124
WHERE  103
construct entities  130
construct entity  133
construct name  133
constructors
 for arrays  79
 for complex objects  22, 23
for structures  38
CONTAINS statement  271
continuation
case  7, 11
lines  6
CONTINUE statement  271
control
edit descriptors  198
format  200
statements
 arithmetic IF  324
 assigned GO TO  321
block IF  325
control (continued)
 statements (continued)
 computed GO TO  322
 CONTINUE  271
 DO  280
 DO WHILE  282
END  292
logical IF  326
PAUSE  363
STOP  400
unconditional GO TO  323
transfer of  14
control mask  105
control structures  117
COS
 intrinsic function  498
 specific name  499
COSD
 intrinsic function  499
 specific name  499
COSH
 intrinsic function  499
 specific name  499
COUNT
array intrinsic function  500
CPU_TIME intrinsic function  501
cpu_time_type run-time option  501
CQABS specific name  479
CQCCOS specific name  499
CQEXP specific name  515
CQLOG specific name  543
CQSSIN specific name  588
CQSQR specific name  594
CSHIFT array intrinsic function  502
CSIN specific name  588
CQRST specific name  594
cmtime_ service and utility subroutine  724
CVMGM, CVMGN, CVMGP, CVMGT, CVMGZ intrinsic functions  503
CYCLE statement  272
D
D (double precision) editing  204
D debug lines  6
DABS specific name  479
DACOS specific name  480
DACOSD specific name  481
DASIN specific name  488
DASIND specific name  488
data
edit descriptors  195, 202
objects  15
statement (DATA)  273
type
 derived  28
types
 conversion rules  90
description  15
intrinsic  16
predefined conventions  51
data transfer
asynchronous  183
executing  182
statement
 PRINT  367
 READ  377
data transfer (continued)
 statement (continued)
 WRITE  425
DATAN specific name  490
DATAN2 specific name  492
DATAN2D specific name  493
DATAND specific name  491
date service and utility subroutine  724
DATE_AND_TIME intrinsic subroutine  505
DBLE
 initializing expressions  85
intrinsic function  506
 specific name  507
DBLEQ specific name  507
DCMPLX
 initializing expressions  85
intrinsic function  507
 specific name  508
DCONJG specific name  498
DCOS specific name  499
DCOSD specific name  499
DCOSH specific name  500
DDIM specific name  509
DEALLOCATE statement  276
debug lines  6, 8
declarators
 array  65
scoping level  130
default typing  51
defered-shape arrays  68
defined assignment  147
defined operations  96
defined operators  146
definition status  52
DELM specifier
 of INQUIRE statement  330
 of OPEN statement  355
DERF specific name  513
DERFC specific name  514
derived types
array structure components  77
description  28
determining the type of  33
scalar structure components  34
structure components  29
structure constructor  38
derived-type statement  278
designators
 for array elements  72
for components  34
DEXP specific name  515
DFLOAT specific name  507
digit-string  15
digits  3
DIGITS intrinsic function  508
DIM
 initializing expressions  85
intrinsic function  509
 specific name  509
DIMAG specific name  483
DIMENSION attribute  279
dimension bound expression  63
dimensions of an array  64
DINT specific name  483
DIRECT specifier, of INQUIRE statement  330
directive lines 6

directives
  CACHE_ZERO 465
discussion 433
  ISYNC 466
  LIGHT_SYNC 466
  NEW 450
  PROTECTED STREAM 469

directives
  @PROCESS 453
  #LINE 447
  ASSERT 436
  assertive 436
  BLOCKLOOP 438
  CSCALL 440
  COLLAPSE 441
  EJECT 442
  INCLUDE 442
  INDEPENDENT 444
  loop optimization 436
  LOOPID 449
  NOSIMD 451
  NOVECTOR 451
  optimization 436
  PERMUTATION 452
  SNAPSHOT 454
  SOURCEFORM 455
  STREAM_UNROLL 456
  SUBSCRIPTORDER 458
  UNROLL 460
  UNROLL_AND_FUSE 461
disconnection, closing files and 181
disjunction, logical 92
division arithmetic operator 89
  DLGAMA specific name 538
  DLOG specific name 543
  DLOG10 specific name 544
  DMAXI specific name 548
  DMIN1 specific name 553
  DMOD specific name 557
  DNIINT specific name 486
  DO
    loop 119, 280, 281
    statement 119, 280
  DO WHILE
    construct 123
    loop 282
    statement 282
dollar ($) editing 219
  DONE specifier, of WAIT statement 421
  DOT_PRODUCT array intrinsic function 509
  DOUBLE COMPLEX type declaration
    statement 283
double precision (D) editing 204
  DOUBLE PRECISION type declaration
    statement 286
double quotation mark (""") editing 199
  DPROD
    initializing expressions 85
    intrinsic function 510
    specific name 510
  DREAL specific name 578
  DSIGN specific name 586
  DSIN specific name 588
  DSIND specific name 589
  DSINH specific name 589
  DSQRT specific name 594
  DTAN specific name 599
  DTAND specific name 599
  DTANH specific name 600
dummy argument
  asterisk as 168
definition of 761
  description 159
  intent attribute and 162
  procedure as 167
  procedure pointer as 167
  variable as 164
dummy procedure 167
dummy procedure pointer 167
dynamic extent, definition of 761

E
E (real with exponent) editing 204
EBCDIC character set 747
edit descriptors
  character string 199
  control (nonrepeatable) 198
data (repeatable) 195
  names and 4
  editing
    \ (colon) 219
    / (slash) 218
    ' (apostrophe) 199
    " (double quotation mark) 199
  $ (dollar) 219
  A (character) 202
  B (binary) 203
  BN (blank null) 220
  BZ (blank zero) 220
  character count Q 215
  character-string 199
  D (double precision) 204
  E (real with exponent) 204
  EN 206
  ES 207
  F (real without exponent) 208
  G (general) 210
  H 211
  I (integer) 212
  L (logical) 213
  O (octal) 214
  P (scale factor) 221
  Q (extended precision) 204
  S, SS, and SP (sign control) 221
  T, TL, TR, and X (positional) 222
  Z (hexadecimal) 217
  efficient floating-point control and
    inquiry procedures
    char_fipscr_flags 693
    discussion 692
    get_fipscr 694
    get_fipscr_flags 694
    get_round_mode 695
    set_fipscr 695
    set_fipscr_flags 695
    set_round_mode 696
  EIEIO compiler directive 465
  EJECT 442
  ELEMENTAL 174
  elemental intrinsic procedures 473
  elemental procedures 174
  ELSE
    block 124
    statement 124, 289
  ELSE IF
    block 124
  ELSEWHERE statement 103, 290
  EN editing 206
  END ASSOCIATE statement 293
  END DO statement 119, 293
  END ENUM statement 303
  END FORALL statement 293
  END IF statement 124, 293
  END INTERFACE statement 141, 296
  END SELECT statement 293
  END specifier
    of READ statement 378
    of WAIT statement 421
  END statement 292
  END TYPE statement 297
  END WHERE statement 103, 293
  end-of-file condition 187
  end-of-record conditions 187
  end-of-record, preventing with $ editing 219
  ENDFILE statement 298
  entities, scope of 130
  entry
    association 319
    name 300
    statement (ENTRY) 300
  ENUM statement 303
  enumerators 303
  EOR specifier, of READ statement 378
  EOSSHIFT array intrinsic function 511
  EPSILON intrinsic function 512
  equivalence
    logical 92
  EQUIVALENCE
    association 305
    restriction on COMMON and 265
  EQUIVALENCE statement 305
  EQV logical operator 92
  ERF
    intrinsic function 513
    specific name 513
  ERFC
    intrinsic function 514
    specific name 514
  ERR specifier
    of BACKSPACE statement 246
    of CLOSE statement 261
    of ENDFILE statement 298
    of INQUIRE statement 330
    of OPEN statement 355
    of READ statement 378
    of REWIND statement 392
    of WAIT statement 421
    of WRITE statement 425
  ERR_RECOVERY run-time option
    BACKSPACE statement and 247
    conversion errors and 191
    EDNFILE statement and 299
    Fortran 90 language errors and 192
    Fortran 95 language errors and 192
ERR_RECOVERY run-time option (continued)
OPEN statement and 399
READ statement and 384
REWIND statement and 393
WRITE statement and 430
error conditions 188
ERROR_UNIT 687
errors catastrophic 188
conversion 191
Fortran 90 language 192
Fortran 95 language 192
recoverable 189
severe 188
ES editing 207
escape sequences 26
etime_service and utility subprogram 725
exclusive disjunction, logical 92
executable program 138
executing data transfer statements 182
execution sequence 14
execution_part 148
EXIST specifier, of INQUIRE statement 330
EXIT statement 307
exit_service and utility subprogram 726
EXP intrinsic function 514
specific name 515
explicit interface 140
typing 51
explicit-shape arrays 66
EXONENT intrinsic function 515
exponentiation arithmetic operator 89
expressions arithmetic 88
character 91
constant 84
dimension bound 63
general 91
in FORMAT statement 317
initialization 85
logical 92
primary 94
relational 95
restricted 86
specification 86
subscript 73
extended intrinsic operations 96
precision (Q) editing 204
external function 317
subprograms in the XL Fortran library 721
EXTERNAL attribute 308
external files 178

F
F (real without exponent) editing 208
factor arithmetic 88
factor (continued)
logical 92
FCFI PowerPC intrinsic function 607
FCFTID PowerPC intrinsic function 607
FCFTIDZ PowerPC intrinsic function 608
FCFTIW PowerPC intrinsic function 608
FCFTIWZ PowerPC intrinsic function 609
fdate_service and utility subprogram 726
fexp.h include file 587
file position
BACKSPACE statement, after execution 247
before and after data transfer 185
ENDFILE statement, after execution 299
REWIND statement, after execution 393
file positioning statement
BACKSPACE statement 246
ENDFILE statement 298
REWIND statement 392
FILE specifier
of INQUIRE statement 330
of OPEN statement 355
FILE_STORAGE_SIZE 687
files 178
fiosetup_service and utility subprogram 726
fixed source form 6
FLOAT specific name 578
FLOOR intrinsic function 516
flush_service and utility subprogram 727
FMADD PowerPC intrinsic function 609
FMSUB PowerPC intrinsic function 610
FMT specifier
of PRINT statement 368
of READ statement 378
of WRITE statement 425
FNABS PowerPC intrinsic function 610
FNMADD PowerPC intrinsic function 611
FNMSUB PowerPC intrinsic function 611
FORALL construct 109
statement 311
FORALL (Construct) statement 314
FORM specifier
of INQUIRE statement 330
of OPEN statement 355
format conditional compilation 11
control 200
fixed source form 6
free source form 9
IBM free source form 11
interaction with input/output list 200
specification character 316
statement (FORMAT) 315
formatted format-directed formatting 195
formatted specifier of INQUIRE statement (FORMATTED) 330
fpgets and fpsets service and utility subprograms 691
fpscr constants
Exception Details Flags 693
Exception Summary Flags 693
general 692
IEEE Exception Enable Flags 693
IEEE Exception Status Flags 693
IEEE Rounding Modes 692
list 692
fpscr procedures
clr_fpscr_flags 693
discussion 692
get_fpscr 694
going_inputs_694
get_round_mode 695
set_fpscr 695
set_fpscr_flags 695
set_round_mode 696
FRAC intrinsic function 517
FRE PowerPC intrinsic function 612
free source form 9
free source form format
IBM 11
FRES PowerPC intrinsic function 612
FRIM PowerPC intrinsic function 613
FRIN PowerPC intrinsic function 613
FRIP PowerPC intrinsic function 614
FRIZ PowerPC intrinsic function 614
FRSQRTE PowerPC intrinsic function 615
FRSQRTES PowerPC intrinsic function 615
FSEL PowerPC intrinsic function 615
ftell_service and utility subprograms 728
ftell64_service and utility subprogram 728
function
intrinsic 721
reference 155
specification 87
statement 397
subprogram 155
value 155
FUNCTION statement 317
G
G (general editing) 210
GAMMA
intrinsic function 517
specific name 518
general expression 91
general service and utility procedures 721
get_fpscr subprogram 694
going_inputs_694
GET_COMMAND intrinsic subroutine 519
GET_COMMAND_ARGUMENT intrinsic subroutine 520
GET_ENVIRONMENT_VARIABLE intrinsic subroutine 521
get_round_mode subprogram 695
getarg service and utility subprogram 729

Index
internal function 317
procedures 138
internal files 178
intrinsic assignment 100
attribute (INTRINSIC) 344
data types 16
functions
See also intrinsic procedures conditional vector merge 503
detailed descriptions 477
generic 156
specific 156
inquiry
See inquiry intrinsic functions procedures 156
description 478
discussion 473
elemental 473
inquiry 473, 474
name in an INTRINSIC statement 345
subroutines 475
transformational 474
statement (INTRINSIC) 145
subroutines 475
invocation commands 6
IOMSG specifier
of BACKSPACE statement 246
of CLOSE statement 261
of ENDFILE statement 298
of INQUIRE statement 330
of OPEN statement 355
of READ statement 378
of REWIND statement 392
of WAIT statement 421
of WRITE statement 425
IOR
intrinsic function 532
specific name 533
IOSTAT specifier
of BACKSPACE statement 246
of CLOSE statement 261
of ENDFILE statement 298
of INQUIRE statement 330
of OPEN statement 355
of READ statement 378
of REWIND statement 392
of WAIT statement 421
of WRITE statement 425
IOSTAT values 186
IOSTAT_END 688
IOSTAT_END FORTRAN 688
IQNINT specific name 531
IQNINT specific name 561
irand service and utility subprogram 734
irtc service and utility subprogram 734
ISHFT
intrinsic function 533
specific name 533
ISHFTC
intrinsic function 534
specific name 534
ISIGN specific name 586
ISO_FORTRAN_ENV intrinsic module 687
ISYNC compiler directive 466
iteration count
DO statement and 120
in implied-DO list of a DATA statement 275
itower_service and utility subprogram 734
J
jdate service and utility subprogram 735
K
keywords
argument 158
statement 5
KIND
intrinsic function 535
intrinsic, constant expressions and 84
intrinsic, restricted expressions 86
kind type parameter 15
L
L (logical) editing 213
labels, statement 5
langlvl run-time option 229
LANGlvl run-time option 192
LBOUND array intrinsic function 535
LEADZ intrinsic function 536
LEN
intrinsic function 537
intrinsic, constant expressions and 84
intrinsic, restricted expressions 86
specific name 537
LEN_TRIM intrinsic function 537
lenchr_service and utility subprogram 735
length type parameter 15
length, inherited by a named constant 260, 413
letters, character 3
lexical tokens 3
lexical extent, definition of 763
LGAMMA
intrinsic function 538
specific name 538
LGE
intrinsic function 539
specific name 539
LGT
intrinsic function 539
specific name 540
library subprograms 721
LIGHT_SYNC compiler directive 466
line breaks, preventing with $ editing 219
lines
comment 6
conditional compilation 11
continuation 6
debug 6, 8
lines (continued)
directive 6, 433
initial 6
source formats and 5
list-directed formatting 223
value separators 223
list-directed input 223
end-of-record 224
rules 224
list-directed output 225
rules 225
types 225
written field width 226
literal storage class 59
LLE
intrinsic function 540
specific name 541
LLT
intrinsic function 541
specific name 541
lnblink_service and utility subprogram 735
LOC
intrinsic function 115, 542
local entities 130, 131
LOG intrinsic function 542
LOG10 intrinsic function 543
logical
(L) editing 213
conjunction 92
data type 24
equivalence 92
exclusive disjunction 92
expressions 92
IP statement 326
inclusive disjunction 92
intrinsic function (LOGICAL) 544
negation 92
nonequivalence 92
type declaration statement (LOGICAL) 346
loop
carried dependency 437, 444
control processing 121
DO construct and 119
LOOPID 449
LSHIFT
elemental function 544
specific name 545
itower_service and utility subprogram 736
M
macro _OPENMP C preprocessor 12
main program 148, 374
many-one section 77
masked array assignment 105
masked ELSEWHERE statement 103, 290
MATMUL array intrinsic function 545
MAX
initializing expressions 85
intrinsic function 547
MAX0 specific name 548
MAX1 specific name 548
MAX_EXPONENT intrinsic function 548
Index 773
N
name
  common block 262
description 4
determining storage class of 59
determining type of 51
type 300
of a generic or specific function 156
scope of a 130
NAME specifier, of INQUIRE statement 330
name-value subsequences 228
named common block 263
NAMED specifier, of INQUIRE statement 330
namelist
  group 4
NAMELIST run-time option 232
statement 353
namelist comments 227
namelist formatting 226
namelist input 227
rules 229
namelist output 231
NEAREST intrinsic function 559
negation
  arithmetic operator 89
  logical operator 92
NEQV logical operator 92
NEW compiler directive 450
NEWLINE
  intrinsic function 560
NEXTREC specifier of INQUIRE statement 330

O
O (octal) editing 214
objects, data 15
octal (O) editing 214
octal constants 47
ONLY clause of USE statement 415
OPEN statement 355
OPENCED specifier, of INQUIRE statement 330
operations
  defined 96
  extended intrinsic 96
operators
  arithmetic 89
  character 91
  defined 146
  logical 92
  precedence of 97
  relational 95
  optional arguments 163
  OPTIONAL attribute 360
OR
  logical operator 92
  specific name 533
order
  of elements in an array 73
  of statements 14
  OUTPUT_UNIT 689

P
P (scale factor) editing 221
PACK array intrinsic function 565
PAD specifier
  of INQUIRE statement 330
  of OPEN statement 355
PARAMETER attribute 362
PAUSE statement 363
pending control mask 105
PERMUTATION 452
Pixel data type 21
pointer
  arrays 67
  POINTER statement and 366
  pointer
  assignment 112
  association 136
  attribute, POINTER (Fortran 90) 364
POPCNT intrinsic function 566
POPCNTB intrinsic function 617
POPPAR intrinsic function 567
POSITION specifier
  of INQUIRE statement 330
  of OPEN statement 355
  positional (T, TL, TR, and X) editing 222
precedence
  of all operators 97
  of arithmetic operators 89
  of logical operators 92
PRECISION intrinsic function 568
precision of real objects 19
preconnection 180
PREFETCH compiler directives 466
PRESENT intrinsic function 361, 569
primaries (expressions) 84
primary expressions 94
PRINT statement 367
PRIVATE attribute 369
statement 30, 369
procedure
dummy 167
external 138, 374
internal 138
procedure pointer, assignment 114
procedure references 155
PROCEDURE statement 371
procedure, invoked by a
subprogram 138
PROCESSORS_SHAPE intrinsic function 569
PRODUCT array intrinsic function 570
PROGRAM statement 374
program unit 138
PROTECTED attribute 374
PROTECTED STREAM directives 469
PUBLIC attribute 376
PURE 171
pure procedures 171

Q
Q (extended precision) editing 204
QABS specific name 479
QACOS specific name 480
QACOSD specific name 481
QARCOS specific name 480
QARSIN specific name 488
QASIN specific name 488
QASIND specific name 488
QATAN specific name 490
QATAN2 specific name 492
QATAN2D specific name 493
QATAND specific name 491
QCMLPX
initializing expressions 85
intrinsic function 572
QCONJG specific name 498
QCOS specific name 499
QCOSD specific name 499
QCOSH specific name 500
QDIM specific name 509
QERF specific name 513
QERFC specific name 514
QEXP specific name 515
QEXT
initializing expressions 85
intrinsic function 572
specific name 573
QEXTD specific name 573
QFLOAT specific name 573
QGAMMA specific name 518
QIMAG specific name 483
QINT specific name 483
QLGAMA specific name 538
QLOG specific name 543
QLOG10 specific name 544
QMAXI specific name 548
QMIN1 specific name 553
QMOD specific name 557
QNINT specific name 486
QPPOD specific name 510
QREAL specific name 578
QSIGN specific name 586
QSN specific name 588
QSN1 specific name 589
QSNH specific name 589
qsort, service and utility
subprogram 737
qsort_down service and utility
subprogram 737
qsort_up service and utility
subprogram 738
QSORT specific name 594
QTAN specific name 599
QTAND specific name 599
QTANH specific name 600
REAL
initializing expressions 85
intrinsic function 577
specific name 578
real data type 18
real editing
E (with exponent) 204
F (without exponent) 208
G (general) 210
REAL type declaration statement 385
REC specifier
of READ statement 378
of WRITE statement 425
RECL specifier
of INQUIRE statement 330
of OPEN statement 355
record statements
statement label (RECORD) 389
RECORD statement 389
records
description 177
recession
ENTRY statement and 302
FUNCTION statement and 320
procedures and 171
SUBROUTINE statement and 402
RECURSIVE keyword 320, 402
reference, function 155
relational
epressions 95
operators 95
REPEAT
intrinsic function 86, 578
intrinsic initialization expressions 85
repeat specification 315
RESHAPE
array intrinsic function 86, 579
array intrinsic initialization
expressions 85
restricted expression 86
RESULT keyword 301, 318
result variable 301, 318
return points and specifiers,
alternate 158
return specifier 14
RETURN statement 390
REWIND statement 392
right margin 7
ROTAELI PowerPC intrinsic
function 618
ROTAELM PowerPC intrinsic
function 619
rounding mode 90
RSRSPACING intrinsic function 580
RSFLICT
elemental function 580
specific name 581
rtc service and utility subprogram 739
run-time options
changing with SETRTEOPTS
procedure 739
CNVRRR
conversion errors and 191
READ statement and 384
WRITE statement and 430
ERR_RECOVERY 192
run-time options (continued)
BACKSPACE statement and 247
conversion errors and 191
ENDFILE statement and 299
OPEN statement and 359
READ statement and 384
REWIND statement and 379
severe errors and 188
WRITE statement and 430
langlvl 229
LANG_LVL 192
NAMELIST 232
NLWIDTH 233
UNIT_VARS 181, 355

S
S (sign control) editing 221
SAVE attribute 393
scalar-int-constant-name 15
scale factor (P) editing 221
SCALE intrinsic function 581
SCAN
initializing expressions 85
intrinsic function 582
intrinsic initialization expressions 85
SELECTED_INT_KIND
intrinsic function 86, 582
intrinsic initialization expressions 85
SELECTED_REAL_KIND
intrinsic function 86, 583
intrinsic initialization expressions 85
selector 4
semicolon statement separator 8, 10
sequence derived type 29
SEQUENCE statement 30, 396
sequential access 178
SEQUENTIAL specifier, of INQUIRE
statement 330
service and utility subprograms
alarm__ 722
bic__ 723
bis__ 723
bit__ 723
clock__ 724
ctime__ 724
date 724
discussion 721
dtime__ 725
efficient floating-point control and
inquiry procedures 692
etime__ 725
exit__ 726
ftime__ 726
fissetup__ 726
flush__ 727
fpgets and fpsets 691
ftell__ 728
ftruncate__ 728
general 721
service and utility subprograms (continued)
getarg 729
getcwd_ 729
getf 730
getgid_ 730
getlog_ 730
getpid_ 731
getuid_ 731
global_timef 731
gmtime_ 732
hostnm_ 732
iargc 733
idate_ 733
iermo_ 733
irand 734
irtc 734
itime_ 734
jd 735
lencr_ 735
lnbkln_ 735
ltime_ 736
mclock 736
qsort 737
qsort_down 737
qsort_up 738
rtc 739
setreopts 739
sleep_ 739
time_ 739
timef 740	timef_delta 740
umask_ 741
usleep_ 741
xl_trbk 741
SET_EXPONENT intrinsic function 584
set_fpscr subprogram 695
set_fpscr_flags subprogram 695
set_round_mode subprogram 696
SETFSB0 PowerPC intrinsic function 619
SETFSB1 PowerPC intrinsic function 619
setreopts service and utility subprogram 739
SFTI PowerPC intrinsic function 620
SHAPE array intrinsic function (SHAPE) 585
of an array 64
of array sections 79
SIGN initializing expressions 85
intrinsic function 585
specif name 586
sign control (S, SS, and SP) editing 221
SIGNAL intrinsic subroutine 587
signal.h include file 587
SIN intrinsic function 587
specif name 588
SIND intrinsic function 588
specif name 589
SINH intrinsic function 589
specif name 589
SIZE array intrinsic function 589
specifier, of READ statement 378
SIZEOF intrinsic function 590
slash (/) editing 218
sleep_ service and utility subprogram 739
SNAPSHOT 454
SNGL specific name 578
SNGLQ specific name 578
sorting (qsort, procedure) 737
source file options 447, 453
source formats
conditional compilation 11
fixed source form 6
free source form 9
IBM free source form 11
SOURCEFORM 455
SP (sign control) editing 221
SPACING intrinsic function 592
special characters 3
specification array 65
specification expression 86
specification function 87
specification_part 148
SPREAD array intrinsic function 592
SQR intrinsic function 593
specif name 594
SRAND intrinsic subroutine 594
SS (sign control) editing 221
statements
assignment 100
BIND 247
block 117
description 5
discussion 235
entities 130, 132, 133
function statement 397
label assignment (ASSIGN)
statement 242
label record (RECORD)
statement 389
labels 5
order 14
terminal 119
STATIC attribute 399
STATUS specifier
of CLOSE statement 261
of OPEN statement 355
STOP statement 400
storage
classes for variables
description 59
fundamental 59
literal 59
secondary 59
sequence within common blocks 265
sharing
using common blocks 263
using EQUIVALENCE 305
using integer pointers 137
using pointers 136
STREAM_UNROLL 456
structure
array components 77
components 29
constructor 38
structure (continued)
description 29
scalar components 34
subobjects of variables 16
subprograms
external 138
function 317
external 155
internal 155
internal 138
invocation 138
references 155
service and utility 721
subroutine 155
subroutine functions and 154
intrinsic 721
statement (SUBROUTINE) 401
subscript_triplet, syntax of 75
SUBSCRIPTORDER 458
subscripts 73
substring character 27
ranges
relationship to array sections 77
specifying 74
subtraction arithmetic operator 89
SUM array intrinsic function 595
SWDIV PowerPC intrinsic function 620
SWDIV_NOCHK PowerPC intrinsic function 621
system inquiry intrinsic functions 474
SYSTEM intrinsic subroutine 596
SYSTEM_CLOCK intrinsic subroutine 597
T (positional) editing 222
tabs, formatting 7
TAN intrinsic function 598
specif name 599
TAND intrinsic function 599
specif name 599
TANH intrinsic function 600
specif name 600
TARGET attribute 403
terminal statement 119
thread-safing
of Fortran 90 pointers 364
time zone, setting 505
time_ service and utility subprogram 739
time service and utility subprogram 740
timef_delta service and utility subprogram 740
TINY intrinsic function 600
TL (positional) editing 222
TR (positional) editing 222
TRANSFER intrinsic function
description 601
initialization expressions 85
restricted expressions 86

776 XL Fortran Language Reference
transfer of control
  description 14
  in a DO loop 121
TRANSFER specifier, of INQUIRE statement 330
transformational intrinsic functions 474
TRANSPOSE array intrinsic function 602
TRAP PowerPC intrinsic function 622
TRIM intrinsic function
description 603
  initialization expressions 85
  restricted expressions 86
type declaration 409
  BYTE 249
  CHARACTER 256
  COMPLEX 266
  DOUBLE COMPLEX 283
  DOUBLE PRECISION 286
  INTEGER 336
  LOGICAL 346
  REAL 385
  TYPE 404
  VECTOR 418
type parameters and specifiers 15
type, determining 51
typeless constants
  binary 47
  hexadecimal 46
  Hollerith 48
  octal 47
  using 49
TZ environment variable 505

U
UBOUND array intrinsic function 603
umask_service and utility
  subprogram 741
unambiguous references 144
unary operations 83
unconditional GO TO statement 323
UNFORMATTED specifier
  of INQUIRE statement 330
Unicode characters and filenames
  and character constants 199
  character constants and
  compiler option for 27
  environment variable for 27
  H editing and 211
  Hollerith constants and 48
UNIT specifier
  of BACKSPACE statement 246
  of CLOSE statement 261
  of ENDFILE statement 298
  of INQUIRE statement 330
  of OPEN statement 355
  of READ statement 378
  of REWIND statement 392
  of WRITE statement 425
units, external files reference 180
UNPACK array intrinsic function 604
UNROLL 460
UNROLL_AND_FUSE 461
unsigned data type 21
use association 135, 414
USE statement 414
usleep_service and utility
  subprogram 741

V
VALUE attribute 417
variable
description 15
  format expressions and 317
  vector data type 20
  vector subscripts 76
VECTOR type declaration statement 418
VERIFY
  initializing expressions 85
  intrinsic function 605
VIRTUAL statement 419
VOLATILE attribute 419

W
WAIT statement 421
WHERE
  construct 103
  construct statement 423
  nested in FORALL 111
  statement 103, 423
  where_construct_name 103, 290, 293, 423
  white space 3
  whole array 63
WRITE
  specifier of INQUIRE statement 330
  statement 425

X
X (positional) editing 222
xl_trbk service and utility
  subprogram 741
xlf_fp_util module 692
xlfutility module 721
XOR
  logical operator 92
  specific name 528

Z
Z (hexadecimal) editing 217
ZABS specific name 479
ZCOS specific name 499
ZEXP specific name 515
ZLOG specific name 543
ZSIN specific name 588
ZSQRT specific name 594
Program Number: 5724-M17