NOTE: This Working Document is only available as a PDF file.

This page intentionally left nonblank.
# Contents

Foreword ............................................. xvii

Introduction .......................................... xviii

1 Scope ................................................... 1

2 Normative references .................................. 3

3 Terms and definitions ................................. 5

4 Notation, conformance, and compatibility ............... 27
   4.1 Notation, symbols and abbreviated terms .......... 27
   4.1.1 Syntax rules .................................. 27
   4.1.2 Constraints .................................... 28
   4.1.3 Assumed syntax rules .......................... 28
   4.1.4 Syntax conventions and characteristics ....... 28
   4.1.5 Text conventions .............................. 29
   4.2 Conformance ....................................... 29
   4.3 Compatibility ...................................... 30
   4.3.1 Previous Fortran standards .................... 30
   4.3.2 New intrinsic procedures ...................... 30
   4.3.3 Fortran 2008 compatibility .................... 30
   4.3.4 Fortran 2003 compatibility .................... 31
   4.3.5 Fortran 95 compatibility ...................... 31
   4.3.6 Fortran 90 compatibility ...................... 32
   4.3.7 FORTRAN 77 compatibility .................... 32
   4.4 Deleted and obsolescent features ............... 33
   4.4.1 General ....................................... 33
   4.4.2 Nature of deleted features ................... 33
   4.4.3 Nature of obsolescent features ............... 33

5 Fortran concepts ..................................... 35
   5.1 High level syntax .................................. 35
   5.2 Program unit concepts ............................. 38
   5.2.1 Program units and scoping units ............... 38
   5.2.2 Program ........................................ 38
   5.2.3 Procedure ...................................... 38
   5.2.4 Module .......................................... 39
   5.2.5 Submodule ..................................... 39
   5.3 Execution concepts ................................ 39
   5.3.1 Statement classification ....................... 39
   5.3.2 Statement order ................................ 39
   5.3.3 The END statement ............................. 40
   5.3.4 Program execution .............................. 40
   5.3.5 Execution sequence ............................. 41
   5.3.6 Image execution states ......................... 41
5.3.7 Termination of execution ................................................................. 42
5.4 Data concepts ................................................................................. 42
5.4.1 Type .............................................................................................. 42
5.4.2 Data value ................................................................................... 43
5.4.3 Data entity ................................................................................... 43
5.4.4 Definition of objects and pointers .................................................. 44
5.4.5 Reference ..................................................................................... 44
5.4.6 Array ............................................................................................ 45
5.4.7 Coarray ....................................................................................... 45
5.4.8 Established coarrays ..................................................................... 46
5.4.9 Pointer .......................................................................................... 46
5.4.10 Allocatable variables ................................................................. 46
5.4.11 Storage ....................................................................................... 46
5.5 Fundamental concepts ................................................................. 46
5.5.1 Names and designators ............................................................... 46
5.5.2 Statement keyword ..................................................................... 47
5.5.3 Other keywords .......................................................................... 47
5.5.4 Association .................................................................................. 47
5.5.5 Intrinsic ....................................................................................... 47
5.5.6 Operator ...................................................................................... 47
5.5.7 Companion processors ............................................................... 47
6 Lexical tokens and source form .......................................................... 49
6.1 Processor character set .................................................................. 49
6.1.1 Characters .................................................................................. 49
6.1.2 Letters ........................................................................................ 49
6.1.3 Digits .......................................................................................... 49
6.1.4 Underscore ................................................................................ 49
6.1.5 Special characters ..................................................................... 49
6.1.6 Other characters ........................................................................ 50
6.2 Low-level syntax .......................................................................... 50
6.2.1 Tokens ....................................................................................... 50
6.2.2 Names ........................................................................................ 50
6.2.3 Constants .................................................................................. 51
6.2.4 Operators .................................................................................. 51
6.2.5 Statement labels ........................................................................ 52
6.2.6 Delimiters .................................................................................. 52
6.3 Source form .................................................................................. 53
6.3.1 Program units, statements, and lines .......................................... 53
6.3.2 Free source form ........................................................................ 53
6.3.3 Fixed source form ..................................................................... 54
6.4 Including source text .................................................................... 55
7 Types ............................................................................................... 57
7.1 Characteristics of types ................................................................ 57
7.1.1 The concept of type .................................................................. 57
7.1.2 Type classification ..................................................................... 57
7.1.3 Set of values .............................................................................. 57
7.1.4 Constants .................................................................................. 57
7.1.5 Operations ................................................................................ 57
7.2 Type parameters ........................................................................... 58
7.3 Types, type specifiers, and values ................................................. 58
7.3.1 Relationship of types and values to objects .................................. 58
7.3.2 Type specifiers and type compatibility ...................................... 59
7.4 Intrinsic types ............................................................................... 61
7.4.1 Classification and specification .................................................. 61
### 7.4 Intrinsic operations on intrinsic types
- 7.4.2 Intrinsic operations on intrinsic types ........................................... 61
- 7.4.3 Numeric intrinsic types ........................................................................ 61
- 7.4.4 Character type ....................................................................................... 65
- 7.4.5 Logical type .......................................................................................... 68

### 7.5 Derived types
- 7.5.1 Derived type concepts ........................................................................... 68
- 7.5.2 Derived-type definition .......................................................................... 69
- 7.5.3 Derived-type parameters ....................................................................... 73
- 7.5.4 Components .......................................................................................... 74
- 7.5.5 Type-bound procedures ......................................................................... 81
- 7.5.6 Final subroutines ................................................................................. 83
- 7.5.7 Type extension ....................................................................................... 85
- 7.5.8 Derived-type values ............................................................................... 87
- 7.5.9 Derived-type specifier .......................................................................... 88
- 7.5.10 Construction of derived-type values .................................................... 88
- 7.5.11 Derived-type operations and assignment .......................................... 90

### 7.6 Enumerations and enumerators ............................................................. 91

### 7.7 Binary, octal, and hexadecimal literal constants ..................................... 92

### 7.8 Construction of array values ................................................................. 93

### 8 Attribute declarations and specifications ................................................ 97
- 8.1 Attributes of procedures and data objects .............................................. 97
- 8.2 Type declaration statement ...................................................................... 97
- 8.3 Automatic data objects ............................................................................ 99
- 8.4 Initialization .............................................................................................. 99
- 8.5 Attributes .................................................................................................. 99
- 8.5.1 Attribute specification .......................................................................... 99
- 8.5.2 Accessibility attribute .......................................................................... 100
- 8.5.3 ALLOCATABLE attribute ..................................................................... 100
- 8.5.4 ASYNCHRONOUS attribute ................................................................. 100
- 8.5.5 BIND attribute for data entities ........................................................... 101
- 8.5.6 CODIMENSION attribute .................................................................... 101
- 8.5.7 CONTIGUOUS attribute ....................................................................... 103
- 8.5.8 DIMENSION attribute ......................................................................... 104
- 8.5.9 EXTERNAL attribute .......................................................................... 107
- 8.5.10 INTENT attribute .............................................................................. 107
- 8.5.11 INTRINSIC attribute .......................................................................... 109
- 8.5.12 OPTIONAL attribute .......................................................................... 109
- 8.5.13 PARAMETER attribute ....................................................................... 110
- 8.5.14 POINTER attribute ............................................................................ 110
- 8.5.15 PROTECTED attribute ....................................................................... 111
- 8.5.16 SAVE attribute .................................................................................. 111
- 8.5.17 TARGET attribute ............................................................................... 112
- 8.5.18 VALUE attribute ................................................................................. 112
- 8.5.19 VOLATILE attribute .......................................................................... 112

### 8.6 Attribute specification statements ......................................................... 113
- 8.6.1 Accessibility statement .......................................................................... 113
- 8.6.2 ALLOCATABLE statement .................................................................... 114
- 8.6.3 ASYNCHRONOUS statement ................................................................ 114
- 8.6.4 BIND statement ................................................................................... 114
- 8.6.5 CODIMENSION statement ................................................................... 115
- 8.6.6 CONTIGUOUS statement ..................................................................... 115
- 8.6.7 DATA statement ................................................................................... 115
- 8.6.8 DIMENSION statement ....................................................................... 117
- 8.6.9 INTENT statement ............................................................................... 118
- 8.6.10 OPTIONAL statement ......................................................................... 118
9 Use of data objects ................................................. 131
  9.1 Designator .................................................. 131
  9.2 Variable ................................................... 131
  9.3 Constants .................................................. 132
  9.4 Scalars ..................................................... 132
     9.4.1 Substrings .......................................... 132
     9.4.2 Structure components ............................... 132
     9.4.3 Coincided named objects ......................... 134
     9.4.4 Complex parts ..................................... 134
     9.4.5 Type parameter inquiry ........................... 134
  9.5 Arrays ..................................................... 135
     9.5.1 Order of reference ................................ 135
     9.5.2 Whole arrays ....................................... 135
     9.5.3 Array elements and array sections ............... 135
     9.5.4 Simply contiguous array designators .......... 138
  9.6 Image selectors ........................................... 139
  9.7 Dynamic association ....................................... 140
     9.7.1 ALLOCATE statement ............................... 140
     9.7.2 NULLIFY statement ................................. 144
     9.7.3 DEALLOCATE statement ......................... 144
     9.7.4 STAT= specifier ................................... 146
     9.7.5 ERRMSG= specifier ................................. 147

10 Expressions and assignment .................................... 149
  10.1 Expressions ................................................ 149
     10.1.1 Expression semantics ............................... 149
     10.1.2 Form of an expression ............................. 149
     10.1.3 Precedence of operators ........................... 153
     10.1.4 Evaluation of operations ......................... 155
     10.1.5 Intrinsic operations ............................... 155
     10.1.6 Defined operations ................................. 162
     10.1.7 Evaluation of operands ......................... 163
     10.1.8 Integrity of parentheses ......................... 164
     10.1.9 Type, type parameters, and shape of an expression 164
     10.1.10 Conformability rules for elemental operations 166
     10.1.11 Specification expression ....................... 166
     10.1.12 Constant expression ......................... 167
  10.2 Assignment ................................................ 169
     10.2.1 Assignment statement ............................. 169
     10.2.2 Pointer assignment ............................... 174
     10.2.3 Masked array assignment – WHERE ................ 177
12.6 Data transfer statements ........................................... 233
  12.6.1 Form of input and output statements ....................... 233
  12.6.2 Control information list ................................... 234
  12.6.3 Data transfer input/output list ............................ 238
  12.6.4 Execution of a data transfer input/output statement .... 241
  12.6.5 Termination of data transfer statements .................. 251
12.7 Waiting on pending data transfer ................................. 252
  12.7.1 Wait operation ................................................ 252
  12.7.2 WAIT statement ................................................. 252
12.8 File positioning statements ....................................... 253
  12.8.1 Syntax .......................................................... 253
  12.8.2 BACKSPACE statement ....................................... 253
  12.8.3 ENDFILE statement ........................................... 254
  12.8.4 REWIND statement ............................................. 254
12.9 FLUSH statement ................................................... 255
12.10 File inquiry statement ........................................... 255
  12.10.1 Forms of the INQUIRE statement ............................ 255
  12.10.2 Inquiry specifiers ............................................ 256
  12.10.3 Inquire by output list ....................................... 262
12.11 Error, end-of-record, and end-of-file conditions ............. 262
  12.11.1 Occurrence of input/output conditions ..................... 262
  12.11.2 Error conditions and the ERR= specifier ................. 262
  12.11.3 End-of-file condition and the END= specifier ............ 263
  12.11.4 End-of-record condition and the EOR= specifier .......... 263
  12.11.5 IOSTAT= specifier ........................................... 264
  12.11.6 IOMSG= specifier ............................................ 264
12.12 Restrictions on input/output statements ........................ 264
13 Input/output editing .................................................. 267
  13.1 Format specifications ............................................. 267
  13.2 Explicit format specification methods .......................... 267
    13.2.1 FORMAT statement ........................................ 267
    13.2.2 Character format specification ............................ 267
  13.3 Form of a format item list ..................................... 268
    13.3.1 Syntax ...................................................... 268
    13.3.2 Edit descriptors ............................................ 268
    13.3.3 Fields ....................................................... 270
  13.4 Interaction between input/output list and format ............. 270
  13.5 Positioning by format control .................................. 272
  13.6 Decimal symbol ................................................... 272
  13.7 Data edit descriptors ............................................ 272
    13.7.1 Purpose of data edit descriptors ......................... 272
    13.7.2 Numeric editing ............................................. 273
    13.7.3 Logical editing ............................................. 280
    13.7.4 Character editing .......................................... 280
    13.7.5 Generalized editing ........................................ 281
    13.7.6 User-defined derived-type editing ......................... 282
  13.8 Control edit descriptors ....................................... 282
    13.8.1 Position edit descriptors .................................. 282
    13.8.2 Slash editing .............................................. 283
    13.8.3 Colon editing ............................................... 283
    13.8.4 SS, SP, and S editing ..................................... 284
    13.8.5 P editing ................................................... 284
    13.8.6 BN and BZ editing .......................................... 284
18 Interoperability with C

18.1 General

18.2 The ISO__C_BINDING intrinsic module

18.3 Interoperability between Fortran and C entities

18.4 C descriptors

18.5 The source file ISO_Fortran_binding.h

18.6 Restrictions on C descriptors
18.7 Restrictions on formal parameters ......................................................... 507
18.8 Restrictions on lifetimes ........................................................................ 508
18.9 Interoperation with C global variables .................................................. 508
  18.9.1 General .......................................................................................... 508
  18.9.2 Binding labels for common blocks and variables ................................. 509
18.10 Interoperation with C functions ........................................................... 510
  18.10.1 Definition and reference of interoperable procedures ....................... 510
  18.10.2 Binding labels for procedures ......................................................... 510
  18.10.3 Exceptions and IEEE arithmetic procedures ...................................... 511
  18.10.4 Asynchronous communication ....................................................... 511

19 Scope, association, and definition ............................................................. 513
  19.1 Scopes, identifiers, and entities ............................................................. 513
  19.2 Global identifiers .................................................................................. 513
  19.3 Local identifiers ................................................................................... 514
    19.3.1 Classes of local identifiers ............................................................ 514
    19.3.2 Local identifiers that are the same as common block names .......... 515
    19.3.3 Function results ........................................................................... 515
    19.3.4 Components, type parameters, and bindings ................................. 515
    19.3.5 Argument keywords .................................................................... 515
  19.4 Statement and construct entities ............................................................ 516
  19.5 Association ......................................................................................... 517
    19.5.1 Name association ......................................................................... 517
    19.5.2 Pointer association ...................................................................... 521
    19.5.3 Storage association ...................................................................... 524
    19.5.4 Inheritance association ................................................................ 526
    19.5.5 Establishing associations .............................................................. 526
  19.6 Definition and undefinition of variables ............................................... 527
    19.6.1 Definition of objects and subobjects .............................................. 527
    19.6.2 Variables that are always defined .................................................. 527
    19.6.3 Variables that are initially defined .................................................. 528
    19.6.4 Variables that are initially undefined ............................................. 528
    19.6.5 Events that cause variables to become defined ............................... 528
    19.6.6 Events that cause variables to become undefined ......................... 530
    19.6.7 Variable definition context ............................................................ 532
    19.6.8 Pointer association context ......................................................... 532

Annex A (informative) Processor Dependencies .......................................... 535
  A.1 Unspecified Items .................................................................................. 535
  A.2 Processor Dependencies ....................................................................... 535

Annex B (informative) Deleted and obsolescent features ............................ 541
  B.1 Deleted features from Fortran 90 ............................................................ 541
  B.2 Deleted features from Fortran 2008 ....................................................... 542
  B.3 Obsolescent features ............................................................................. 542
    B.3.1 General .......................................................................................... 542
    B.3.2 Alternate return ............................................................................. 542
    B.3.3 Computed GO TO statement .......................................................... 543
    B.3.4 Statement functions ...................................................................... 543
    B.3.5 DATA statements among executables .............................................. 543
    B.3.6 Assumed character length functions .............................................. 543
    B.3.7 Fixed form source ......................................................................... 543
    B.3.8 CHARACTER* form of CHARACTER declaration ......................... 544
    B.3.9 ENTRY statements ....................................................................... 544
    B.3.10 Label DO statement .................................................................... 544
    B.3.11 COMMON and EQUIVALENCE statements and the block data program unit ................................. 544
List of Tables

4.3 Previous editions of the Fortran International Standard ................................ 30
5.1 Requirements on statement ordering .............................................................. 39
5.2 Statements allowed in scoping units ............................................................... 40
6.1 Special characters ............................................................................................ 50
6.2 Adjacent keywords where separating blanks are optional ............................. 53
9.1 Subscript order value ....................................................................................... 136
10.1 Categories of operations and relative precedence ........................................ 153
10.2 Type of operands and results for intrinsic operators ..................................... 156
10.3 Interpretation of the numeric intrinsic operators .......................................... 157
10.4 Interpretation of the character intrinsic operator // ...................................... 159
10.5 Interpretation of the logical intrinsic operators ............................................ 160
10.6 The values of operations involving logical intrinsic operators ...................... 160
10.7 Interpretation of the relational intrinsic operators ......................................... 161
10.8 Type conformance for the intrinsic assignment statement ......................... 169
10.9 Numeric conversion and the assignment statement ...................................... 171
13.1 E and D exponent forms ............................................................................... 276
13.2 EN exponent forms ....................................................................................... 277
13.3 ES exponent forms ....................................................................................... 277
16.1 Standard generic intrinsic procedure summary .......................................... 346
16.2 Unrestricted specific intrinsic functions ...................................................... 352
16.3 Restricted specific intrinsic functions ......................................................... 353
16.4 Default BOUNDARY values for EOSHIFT .............................................. 379
16.5 Characteristics of the result of NULL () ..................................................... 415
17.1 IEEE relational operator correspondence ................................................... 450
17.2 IEEE_ARITHMETIC module procedure summary ..................................... 454
17.3 IEEE_EXCEPTIONS module procedure summary .................................................. 455
18.1 Names of C characters with special semantics .................................................... 484
18.2 Interoperability between Fortran and C types ...................................................... 489
18.3 ISO_Fortran_binding.h macros for attribute codes .............................................. 498
18.4 ISO_Fortran_binding.h macros for type codes ..................................................... 498
18.5 ISO_Fortran_binding.h macros for error codes .................................................... 499
Foreword

1 ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and nongovernmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

2 International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

3 The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75% of the national bodies casting a vote.

4 Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

5 ISO/IEC 1539-1 was prepared by Joint Technical Committee ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming languages, their environments and system software interfaces.


7 ISO/IEC 1539 consists of the following parts, under the general title Information technology — Programming languages — Fortran:

8 — Part 1: Base language

9 — Part 2: Varying length character strings
Introduction

1 This document comprises the specification of the base Fortran language, informally known as Fortran 2015. With the limitations noted in 4.3.4, the syntax and semantics of Fortran 2008 are contained entirely within Fortran 2015. Therefore, any standard-conforming Fortran 2008 program not affected by such limitations is a standard-conforming Fortran 2015 program. New features of Fortran 2015 can be compatibly incorporated into such Fortran 2008 programs, with any exceptions indicated in the text of this document.

2 Fortran 2015 contains several extensions to Fortran 2008; these are listed below.

- **Data declaration:**
  Constant properties of an object declared in its `entity-decl` can be used in its `initialization`. The `EQUIVALENCE` and `COMMON` statements and the block data program unit have been redundant since Fortran 90 and are now specified to be obsolete.

- **Data usage and computation:**
  The declared type of the value supplied for a polymorphic allocatable component in a structure constructor is no longer required to be the same as the declared type of the component. `FORALL` is now specified to be obsolete. The type and kind of an implied DO variable in an array constructor or `DATA` statement can be specified within the constructor or statement. The `SELECT RANK` construct provides structured access to the elements of an assumed-rank array. Completing execution of a `BLOCK` construct can cause the association status of a pointer with the `PROTECTED` attribute to become undefined. The standard intrinsic operations `<`, `<=`, `>`, and `>=` (also known as `.LT.`, `.LE.`, `.GT.`, and `.GE.`) on IEEE numbers provide compareSignaling{relation} operations; the `=` and `/=` operations (also known as `.EQ.` and `.NE.`) provide compareQuiet{relation} operations. Finalization of an allocatable subobject during intrinsic assignment has been clarified.

- **Input/output:**
  The `SIZE=` specifier can be used with advancing input. It is no longer prohibited to open a file on more than one unit. The value assigned by the `RECL=` specifier in an `INQUIRE` statement has been standardized. The values assigned by the `POS=` and `SIZE=` specifiers in an `INQUIRE` statement for a unit that has pending asynchronous operations have been standardized. The `G0.d edit descriptor can be used for list items of type `Integer`, `Logical`, and `Character`. The `D`, `E`, `EN`, and `ES` edit descriptors can have a field width of zero, analogous to the `F` edit descriptor. The exponent width `e` in a `data edit descriptor` can be zero, analogous to a field width of zero. Floating-point formatted input accepts hexadecimal-significand numbers that conform to ISO/IEC/IEEE 60559:2011. The `EX edit descriptor` provides hexadecimal-significand formatted output conforming to ISO/IEC/IEEE 60559:2011. An error condition occurs if unacceptable characters are presented for logical or numeric editing during execution of a formatted input statement.

- **Execution control:**
  The `arithmetic IF statement` has been deleted. Labeled DO loops have been redundant since Fortran 90 and are now specified to be obsolete. The `nonblock DO construct` has been deleted. The locality of a variable used in a `DO CONCURRENT` construct can be explicitly specified. The stop code in a `STOP` or `ERROR STOP` statement can be nonconstant. Output of the stop code and exception summary from the `STOP` and `ERROR STOP` statements can be controlled.

- **Intrinsic procedures and modules:**
  In references to the intrinsic functions `ALL`, `ANY`, `FINDLOC`, `IALL`, `IANY`, `IPARITY`, `MAXLOC`, `MAXVAL`, `MINLOC`, `MINVAL`, `NORM2`, `PARITY`, `PRODUCT`, `SUM`, and `THIS_IMAGE`, the actual argument for `DIM` can be a present optional dummy argument. In a reference to the intrinsic function `CMPLX` with an actual argument of type complex, no keyword is needed for a KIND argument. The new intrinsic function `COSHAPE` returns the coshape of a `coarray`. The new intrinsic function `OUT_OF_RANGE` tests whether a numeric value can be safely converted to a different type or kind. The new intrinsic subroutine `RANDOM_` - `INIT` establishes the initial state of the pseudorandom number generator used by `RANDOM_NUMBER`. The new intrinsic function `REDUCE` performs user-specified array reductions. A processor is required to report use of a nonstandard intrinsic procedure, use of a nonstandard intrinsic module, and use of a nonstandard procedure from a standard intrinsic module. Integer and logical arguments to intrinsic procedures and intrinsic module procedures that were previously required to be of default kind no longer have that requirement, except for `RANDOM_SEED`. Specific names for intrinsic functions are now deemed obsolete.
All standard procedures in the intrinsic module ISO_C_BINDING, other than C_F_POINTER, are now pure. The arguments to the intrinsic function SIGN can be of different kind. Nonpolymorphic pointer arguments to the intrinsic functions EXTENDS_TYPE_OF and SAME_TYPE_AS need not have defined pointer association status. The effects of invoking the intrinsic procedures COMMAND_ARGUMENT_COUNT, GET_COMMAND, and GET_COMMAND_ARGUMENT, on images other than image one, are no longer processor dependent. Access to error messages from the intrinsic subroutines GET_COMMAND, GET_COMMAND_ARGUMENT, and GET_ENVIRONMENT_VARIABLE is provided by an optional ERRMSG argument.

Program units and procedures:
The IMPORT statement can appear in a contained subprogram or BLOCK construct, and can restrict access via host association; diagnosis of violation of the IMPORT restrictions is required. The GENERIC statement can be used to declare generic interfaces. The number of procedure arguments is used in generic resolution. In a module, the default accessibility of entities accessed from another module can be controlled separately from the default accessibility of entities declared in the using module. An IMPLICIT NONE statement can require explicit declaration of the EXTERNAL attribute throughout a scoping unit and its contained scoping units. A defined operation need not specify INTENT (IN) for a dummy argument with the VALUE attribute. A defined assignment need not specify INTENT (IN) for the second dummy argument if it has the VALUE attribute. Procedures, including elemental procedures, can be invoked recursively by default; the NON_RECURSIVE keyword is advisory only. The character length of a dummy data object of an interoperable procedure can be assumed. The argument to C_LOC can be a noninteroperable array. The FPTR argument to C_F_POINTER can be a noninteroperable array pointer. The argument to C_FUNLOC can be a noninteroperable procedure. The FPTR argument to C_F_PROCPOINTER can be a noninteroperable procedure pointer.

Additionally to ISO/IEC TS 29113:2012, an assumed-rank dummy data object that is not associated with an assumed-size array can be used as the argument to the function C_SIZEOF from the intrinsic module ISO_C_BINDING.

Changes to the intrinsic modules IEEE_ARITHMETIC, IEEE_EXCEPTIONS, and IEEE_FEATURES for conformance with ISO/IEC/IEEE 60559:2011:
There is a new, optional, rounding mode IEEE_AWAY. The new type IEEE_MODES_TYPE encapsulates all floating-point modes. Features associated with subnormal numbers can be accessed with functions and types named ...SUBNORMAL... (the old ...DENORMAL... names remain). The new function IEEE_FMA performs fused multiply-add operations. The function IEEE_INT performs rounded conversions to integer type. The new functions IEEE_MAX_NUM, IEEE_MAX_NUM_MAG, IEEE_MIN_-NUM, and IEEE_MIN_NUM_MAG calculate maximum and minimum numeric values. The new functions IEEE_NEXT_DOWN and IEEE_NEXT_UP return the adjacent machine numbers. The new functions IEEE_QIUIT_EQ, IEEE_QIUIT_GE, IEEE_QIUIT_GT, IEEE_QIUIT_LE, IEEE_QIUIT_LT, and IEEE_QIUIT_NE perform quiet comparisons. The new functions IEEE_SIGNALING_EQ, IEEE_SIGNALING_GE, IEEE_SIGNALING_GT, IEEE_SIGNALING_GE, IEEE_SIGNALING_LE, IEEE_SIGNALING_LT, and IEEE_SIGNALING_NE perform signaling comparisons. The decimal rounding mode can be inquired and set independently of the binary rounding mode, using the RADIX argument to IEEE_GET_ROUNDING_MODE and IEEE_SET_ROUNDING_MODE. The new function IEEE_SIGNBIT tests the sign bit of an IEEE number.

Features previously described by ISO/IEC TS 18508:2015:
The CRITICAL statement has optional ERRMSG= and STAT= specifiers. The intrinsic subroutines ATOMIC_DEFINE and ATOMIC_REF have an optional STAT argument. The new intrinsic subroutines
ATOMIC_ADD, ATOMIC_AND, ATOMIC_CAS, ATOMIC_FETCH_ADD, ATOMIC_FETCH_AND, ATOMIC_FETCH_OR, ATOMIC_FETCH_XOR, ATOMIC_OR, and ATOMIC_XOR perform atomic operations. The new intrinsic functions FAILED_IMAGES and STOPPED_IMAGES return indices of images known to have failed or stopped respectively. The new intrinsic function IMAGE_STATUS returns the image execution status of an image. The intrinsic subroutine MOVE_ALLOC has optional ERRMSG and STAT arguments. The intrinsic functions IMAGE_INDEX and NUM_IMAGES have additional forms with a TEAM or TEAM_NUMBER argument. The intrinsic function THIS_IMAGE has an optional TEAM argument. The EVENT POST and EVENT WAIT statements, the intrinsic subroutine EVENT__QUERY, and the type EVENT_TYPE provide an event facility for one-sided segment ordering. The CHANGE TEAM construct, derived type TEAM_TYPE, FORM TEAM and SYNC TEAM statements, the intrinsic functions GET_TEAM and TEAM_NUMBER, and the STAT= and TEAM= specifiers on image selectors, provide a team facility for a subset of the program’s images to act in concert as if it were the set of all images. The new intrinsic subroutines CO_BROADCAST, CO_MAX, CO_MIN, CO_REDUCE, and CO_SUM perform collective reduction operations on the images of the current team. The concept of failed images, the FAIL IMAGE statement, and the named constant STAT_FAILED_IMAGE provide support for fault-tolerant parallel execution.

This document is organized in 16 clauses, dealing with 8 conceptual areas. These 8 areas, and the clauses in which they are treated, are:

<table>
<thead>
<tr>
<th>Conceptual Area</th>
<th>Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>High/low level concepts</td>
<td>4, 5, 6</td>
</tr>
<tr>
<td>Data concepts</td>
<td>7, 8, 9</td>
</tr>
<tr>
<td>Computations</td>
<td>10, 16, 17</td>
</tr>
<tr>
<td>Execution control</td>
<td>11</td>
</tr>
<tr>
<td>Input/output</td>
<td>12, 13</td>
</tr>
<tr>
<td>Program units</td>
<td>14, 15</td>
</tr>
<tr>
<td>Interoperability with C</td>
<td>18</td>
</tr>
<tr>
<td>Scoping and association rules</td>
<td>19</td>
</tr>
</tbody>
</table>

It also contains the following nonnormative material:

<table>
<thead>
<tr>
<th>Material</th>
<th>Annex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor dependencies</td>
<td>A</td>
</tr>
<tr>
<td>Deleted and obsolescent features</td>
<td>B</td>
</tr>
<tr>
<td>Extended notes</td>
<td>C</td>
</tr>
<tr>
<td>Index</td>
<td></td>
</tr>
</tbody>
</table>
Information technology — Programming languages — Fortran —

Part 1:
Base language

1 Scope

1 This document specifies the form and establishes the interpretation of programs expressed in the base Fortran language. The purpose of this document is to promote portability, reliability, maintainability, and efficient execution of Fortran programs for use on a variety of computing systems.

2 This document specifies

• the forms that a program written in the Fortran language may take,
• the rules for interpreting the meaning of a program and its data,
• the form of the input data to be processed by such a program, and
• the form of the output data resulting from the use of such a program.

3 Except where stated otherwise, requirements and prohibitions specified by this document apply to programs rather than processors.

4 This document does not specify

• the mechanism by which programs are transformed for use on computing systems,
• the operations required for setup and control of the use of programs on computing systems,
• the method of transcription of programs or their input or output data to or from a storage medium,
• the program and processor behavior when this document fails to establish an interpretation except for the processor detection and reporting requirements in items (2) to (10) of 4.2,
• the maximum number of images, or the size or complexity of a program and its data that will exceed the capacity of any particular computing system or the capability of a particular processor,
• the mechanism for determining the number of images of a program,
• the physical properties of an image or the relationship between images and the computational elements of a computing system,
• the physical properties of the representation of quantities and the method of rounding, approximating, or computing numeric values on a particular processor, except by reference to ISO/IEC/IEEE 60559:2011 under conditions specified in Clause 17,
• the physical properties of input/output records, files, and units, or
• the physical properties and implementation of storage.
2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes
requirements of this document. For dated references, only the edition cited applies. For undated references, the
latest edition of the referenced document (including any amendments) applies.

information interchange*


ISO/IEC 10646, *Information technology—Universal Multiple-Octet Coded Character Set (UCS)*

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply. ISO and IEC maintain terminological databases for use in standardization at the following addresses:

- ISO Online browsing platform: available at http://www.iso.org/obp

3.1 actual argument
entity (R1524) that appears in a procedure reference

3.2 allocatable
having the ALLOCATABLE attribute (8.5.3)

3.3 array
set of scalar data, all of the same type and type parameters, whose individual elements are arranged in a rectangular pattern (8.5.8, 9.5)

3.3.1 array element
scalar individual element of an array

3.3.2 array pointer
array with the POINTER attribute (8.5.14)

3.3.3 array section
array subobject designated by array-section, and which is itself an array (9.5.3.3)

3.3.4 assumed-shape array
nonallocatable nonpointer dummy argument array that takes its shape from its effective argument (8.5.8.3)

3.3.5 assumed-size array
dummy argument array whose size is assumed from that of its effective argument (8.5.8.5)

3.3.6 deferred-shape array
allocatable array or array pointer, declared with a deferred-shape-spec-list (8.5.8.4)

3.3.7 explicit-shape array
array declared with an explicit-shape-spec-list, which specifies explicit values for the bounds in each dimension of the array (8.5.8.2)

3.4 ASCII character
character whose representation method corresponds to ISO/IEC 646:1991 (International Reference Version)
3.5
associate name
name of construct entity associated with a selector of an ASSOCIATE, CHANGE TEAM, SELECT RANK, or SELECT TYPE construct (11.1.3, 11.1.5, 11.1.10, 11.1.11)

3.6
associating entity
(in a dynamically-established association) the entity that did not exist prior to the establishment of the association (19.5.5)

3.7
association
inheritance association, name association, pointer association, or storage association.

3.7.1
argument association
association between an effective argument and a dummy argument (15.5.2)

3.7.2
construct association
association between a selector and an associate name in an ASSOCIATE, CHANGE TEAM, SELECT RANK, or SELECT TYPE construct(11.1.3, 11.1.5, 11.1.10, 11.1.11, 19.5.1.6)

3.7.3
host association
name association, other than argument association, between entities in a submodule or contained scoping unit and entities in its host (19.5.1.4)

3.7.4
inheritance association
association between the inherited components of an extended type and the components of its parent component (19.5.4)

3.7.5
linkage association
association between a variable or common block with the BIND attribute and a C global variable (18.9, 19.5.1.5)

3.7.6
name association
argument association, construct association, host association, linkage association, or use association (19.5.1)

3.7.7
pointer association
association between a pointer and an entity with the TARGET attribute (19.5.2)

3.7.8
storage association
association between storage sequences (19.5.3)

3.7.9
use association
association between entities in a module and entities in a scoping unit or construct that references that module, as specified by a USE statement (14.2.2)

3.8
assumed-rank dummy data object
dummy data object that assumes the rank, shape, and size of its effective argument (8.5.8.7)
3.9 assumed-type
3 declared with a TYPE(*) type specifier (7.3.2)

3.10 attribute
6 property of an entity that determines its uses (8.1)

3.11 automatic data object
9 nondummy data object with a type parameter or array bound that depends on the value of a specification-expr
that is not a constant expression (8.3)

3.12 base object
13 (data-ref) object designated by the leftmost part-name (9.4.2)

3.13 binding
16 type-bound procedure or final subroutine (7.5.5)

3.14 binding name
19 name given to a specific or generic type-bound procedure in the type definition (7.5.5)

3.15 binding label
22 default character value specifying the name by which a global entity with the BIND attribute is known to the
companion processor (18.10.2, 18.9.2)

3.16 block
26 sequence of executable constructs formed by the syntactic class block and which is treated as a unit by the
executable constructs described in 11.1

3.17 bound
30 array bound
31 limit of a dimension of an array (8.5.8)

3.18 branch target statement
34 action-stmt, associate-stmt, end-associate-stmt, if-then-stmt, end-if-stmt, select-case-stmt, end-select-stmt, select-
rank-stmt, end-select-rank-stmt, select-type-stmt, end-select-type-stmt, do-stmt, end-do-stmt, block-stmt, end-
block-stmt, critical-stmt, end-critical-stmt, forall-construct-stmt, where-construct-stmt, end-function-stmt, end-mp-
subprogram-stmt, end-program-stmt, or end-subroutine-stmt, whose statement label appears as a label in a GO TO
statement, computed GO TO statement, alt-return-spec, END= specifier, EOR= specifier, or ERR= specifier (11.2.1)

3.19 C address
41 value identifying the location of a data object or procedure either defined by the companion processor or which
might be accessible to the companion processor

NOTE 3.1
This is the concept that ISO/IEC 9899:2011 calls the address.
3.20
C descriptor
C structure of type CFI_cdesc_t defined in the source file ISO_Fortran_binding.h (18.4, 18.5)

3.21
character context
within a character literal constant (7.4.4) or within a character string edit descriptor (13.3.2)

3.22
characteristics
⟨dummy argument⟩ being a dummy data object, dummy procedure, or an asterisk (alternate return indicator)

3.23
characteristics
⟨dummy data object⟩ properties listed in 15.3.2.2

3.24
characteristics
⟨dummy procedure or dummy procedure pointer⟩ properties listed in 15.3.2.3

3.25
characteristics
⟨function result⟩ properties listed in 15.3.3

3.26
characteristics
⟨procedure⟩ properties listed in 15.3.1

3.27
coarray
data entity that has nonzero corank (5.4.7)

3.27.1
established coarray
coarray that is accessible using an image-selector (5.4.7)

3.28
cobound
bound (limit) of a codimension (8.5.6)

3.29
codimension
dimension of the pattern formed by a set of corresponding coarrays (8.5.6)

3.30
coindexed object
data object whose designator includes an image-selector (R924, 9.6)

3.31
collating sequence
one-to-one mapping from a character set into the nonnegative integers (7.4.4.4)

3.32
common block
block of physical storage specified by a COMMON statement (8.10.2)

3.32.1
blank common
unnamed common block
3.33 companion processor
processor-dependent mechanism by which global data and procedures may be referenced or defined (5.5.7)

3.34 component
part of a derived type, or of an object of derived type, defined by a component-def-stmt (7.5.4)

3.34.1 direct component
one of the components, or one of the direct components of a nonpointer nonallocatable component (7.5.1)

3.34.2 parent component
component of an extended type whose type is that of the parent type and whose components are inheritance associated with the inherited components of the parent type (7.5.7.2)

3.34.3 potential subobject component
nonpointer component, or potential subobject component of a nonpointer component (7.5.1)

3.34.4 subcomponent
(structure) direct component that is a subobject of the structure (9.4.2)

3.34.5 ultimate component
component that is of intrinsic type, a pointer, or allocatable; or an ultimate component of a nonpointer nonallocatable component of derived type

3.35 component order
ordering of the nonparent components of a derived type that is used for intrinsic formatted input/output and structure constructors (where component keywords are not used) (7.5.4.7)

3.36 conformable
(of two data entities) having the same shape, or one being an array and the other being scalar

3.37 connected
relationship between a unit and a file: each is connected if and only if the unit refers to the file (12.5.4)

3.38 constant
data object that has a value and which cannot be defined, redefined, or become undefined during execution of a program (6.2.3, 9.3)

3.38.1 literal constant
constant that does not have a name (R605, 7.4)

3.38.2 named constant
named data object with the PARAMETER attribute (8.5.13)
3.39
construct entity
entity whose identifier has the scope of a construct (19.1, 19.4)

3.40
constant expression
expression satisfying the requirements specified in 10.1.12, thus ensuring that its value is constant

3.41
contiguous
(array) having array elements in order that are not separated by other data objects, as specified in 8.5.7

3.42
contiguous
(multi-part data object) that the parts in order are not separated by other data objects

3.43
corank
number of codimensions of a coarray (zero for objects that are not coarrays) (8.5.6)

3.44
cosubscript
(R925) scalar integer expression in an image-selector (R924)

3.45
data entity
data object, result of the evaluation of an expression, or the result of the execution of a function reference

3.46
data object
object
constant (7.1.4), variable (9), or subobject of a constant (5.4.3.2.3)

3.47
decimal symbol
character that separates the whole and fractional parts in the decimal representation of a real number in a file (13.6)

3.48
declaration
specification of attributes for various program entities

NOTE 3.2
Often this involves specifying the type of a named data object or specifying the shape of a named array object.

3.49
default initialization
mechanism for automatically initializing pointer components to have a defined pointer association status, and nonpointer components to have a particular value (7.5.4.6)

3.50
default-initialized
(subcomponent) subject to a default initialization specified in the type definition for that component (7.5.4.6)

3.51
definable
capable of definition and permitted to become defined
3.52
defined
⟨data object⟩ has a valid value

3.53
defined
⟨pointer⟩ has a pointer association status of associated or disassociated (19.5.2.2)

3.54
defined assignment
assignment defined by a procedure (10.2.1.4, 15.4.3.4.3)

3.55
defined input/output
input/output defined by a procedure and accessed via a defined-in-generic-spec (R1509, 12.6.4.8)

3.56
defined operation
operation defined by a procedure (10.1.6.1, 15.4.3.4.2)

3.57
definition
⟨data object⟩ process by which the data object becomes defined (19.6.5)

3.58
definition
⟨derived type (7.5.2), enumeration (7.6), or procedure (15.6)⟩ specification of the type, enumeration, or procedure

3.59
descendant
⟨module or submodule⟩ submodule that extends that module or submodule or that extends another descendant thereof (14.2.3)

3.60
designator
name followed by zero or more component selectors, complex part selectors, array section selectors, array element selectors, image selectors, and substring selectors (9.1)

3.60.1
complex part designator
designator that designates the real or imaginary part of a complex data object, independently of the other part (9.4.4)

3.60.2
object designator
data object designator
designator for a data object

NOTE 3.3
An object name is a special case of an object designator.

3.60.3
procedure designator
designator for a procedure
3.61
**disassociated**
(pointer association) pointer association status of not being associated with any target and not being undefined (19.5.2.2)

3.62
**disassociated**
(pointer) has a pointer association status of disassociated

3.63
dummy argument
entity whose identifier appears in a dummy argument list (R1536) in a FUNCTION, SUBROUTINE, ENTRY, or statement function statement, or whose name can be used as an argument keyword in a reference to an intrinsic procedure or a procedure in an intrinsic module

3.63.1
dummy data object
dummy argument that is a data object

3.63.2
dummy function
dummy procedure that is a function

3.64
effective argument
entity that is argument-associated with a dummy argument (15.5.2.3)

3.65
effective item
scalar object resulting from the application of the rules in 12.6.3 to an input/output list

3.66
**elemental**
independent scalar application of an action or operation to elements of an array or corresponding elements of a set of conformable arrays and scalars, or possessing the capability of elemental operation

**NOTE 3.4**
Combination of scalar and array operands or arguments combine the scalar operand(s) with each element of the array operand(s).

3.66.1
elemental assignment
assignment that operates elementally

3.66.2
elemental operation
operation that operates elementally

3.66.3
elemental operator
operator in an elemental operation

3.66.4
elemental procedure
elemental intrinsic procedure or procedure defined by an elemental subprogram (15.8)
3.66.5  
**elemental reference**  
reference to an elemental procedure with at least one array actual argument

3.66.6  
**elemental subprogram**  
subprogram with the **ELEMENTAL** prefix (15.8.1)

3.67  
**END statement**  
`end-block-data-stmt`, `end-function-stmt`, `end-module-stmt`, `end-mp-subprogram-stmt`, `end-program-stmt`, `end-submodule-stmt`, or `end-subroutine-stmt`

3.68  
**explicit initialization**  
initialization of a data object by a specification statement (8.4, 8.6.7)

3.69  
**extent**  
number of elements in a single dimension of an **array**

3.70  
**external file**  
file that exists in a medium external to the program (12.3)

3.71  
**external unit**  
**external input/output unit**  
entity that can be **connected** to an **external file** (12.5.3, 12.5.4)

3.72  
**file storage unit**  
unit of storage in a **stream file** or an unformatted **record file** (12.3.5)

3.73  
**final subroutine**  
subroutine whose name appears in a **FINAL statement** (7.5.6) in a type definition, and which can be automatically invoked by the processor when an object of that type is finalized (7.5.6.2)

3.74  
**finalizable**  
⟨**type**⟩ has a final subroutine or a nonpointer nonallocatable component of finalizable type

3.75  
**finalizable**  
⟨**nonpointer data entity**⟩ of finalizable type

3.76  
**finalization**  
process of calling **final subroutines** when one of the events listed in 7.5.6.3 occurs

3.77  
**function**  
procedure that is invoked by an expression

3.78  
**function result**  
entity that returns the value of a function (15.6.2.2)
3.79
generic identifier
lexical token that identifies a generic set of procedures, intrinsic operations, and/or intrinsic assignments (15.4.3.4.1)

3.80
host instance
(internal procedure, or dummy procedure or procedure pointer associated with an internal procedure) instance of the host procedure that supplies the host environment of the internal procedure (15.6.2.4)

3.81
host scoping unit
host scoping unit immediately surrounding another scoping unit, or the scoping unit extended by a submodule

3.82
IEEE infinity
ISO/IEC/IEEE 60559:2011 conformant infinite floating-point value

3.83
IEEE NaN
ISO/IEC/IEEE 60559:2011 conformant floating-point datum that does not represent a number

3.84
image
instance of a Fortran program (5.3.4)

3.84.1
active image
image that has not failed or stopped (5.3.6)

3.84.2
failed image
image that has not initiated termination but which has ceased to participate in program execution (5.3.6)

3.84.3
stopped image
image that has initiated normal termination (5.3.6)

3.85
image index
integer value identifying an image within a team

3.86
image control statement
statement that affects the execution ordering between images (11.6)

3.87
inclusive scope
nonblock scoping unit plus every block scoping unit whose host is that scoping unit or that is nested within such a block scoping unit

NOTE 3.5
That is, inclusive scope is the scope as if BLOCK constructs were not scoping units.

3.88
inherit
(extended type) acquire entities (components, type-bound procedures, and type parameters) through type extension from the parent type (7.5.7.2)
3.89
inquiry function
intrinsic function, or function in an intrinsic module, whose result depends on the properties of one or more of its arguments instead of their values.

3.90
interface
(procedure) name, procedure characteristics, dummy argument names, binding label, and generic identifiers (15.4.1)

3.90.1
abstract interface
set of procedure characteristics with dummy argument names (15.4.1)

3.90.2
explicit interface
interface of a procedure that includes all the characteristics of the procedure and names for its dummy arguments except for asterisk dummy arguments (15.4.2)

3.90.3
generic interface
set of procedure interfaces identified by a generic identifier

3.90.4
implicit interface
interface of a procedure that is not an explicit interface (15.4.2, 15.4.3.8)

3.90.5
specific interface
interface identified by a nongeneric name

3.91
interface block
abstract interface block, generic interface block, or specific interface block (15.4.3.2)

3.91.1
abstract interface block
interface block with the ABSTRACT keyword; collection of interface bodies that specify named abstract interfaces

3.91.2
generic interface block
interface block with a generic-spec; collection of interface bodies and procedure statements that are to be given that generic identifier

3.91.3
specific interface block
interface block with no generic-spec or ABSTRACT keyword; collection of interface bodies that specify the interfaces of procedures

3.92
interoperable
(Fortran entity) equivalent to an entity defined by or definable by the companion processor (18.3)

3.93
intrinsic
type, procedure, module, assignment, operator, or input/output operation defined in this document and accessible without further definition or specification, or a procedure or module provided by a processor but not defined in this document
3.93.1
standard intrinsic
(procedure or module) defined in this document (16)

3.93.2
nonstandard intrinsic
(procedure or module) provided by a processor but not defined in this document

3.94
internal file
character variable that is connected to an internal unit (12.4)

3.95
internal unit
input/output unit that is connected to an internal file (12.5.4)

3.96
ISO 10646 character
character whose representation method corresponds to UCS-4 in ISO/IEC 10646

3.97
keyword
statement keyword, argument keyword, type parameter keyword, or component keyword

3.97.1
argument keyword
word that identifies the corresponding dummy argument in an actual argument list (15.5.2.1)

3.97.2
component keyword
word that identifies a component in a structure constructor (7.5.10)

3.97.3
statement keyword
word that is part of the syntax of a statement (5.5.2)

3.97.4
type parameter keyword
word that identifies a type parameter in a type parameter list

3.98
lexical token
keyword, name, literal constant other than a complex literal constant, operator, label, delimiter, comma, =, =>, ::, ;, or % (6.2)

3.99
line
sequence of zero or more characters

3.100
main program
program unit that is not a subprogram, module, submodule, or block data program unit (14.1)

3.101
masked array assignment
assignment statement in a WHERE statement or WHERE construct (10.2.3)
module
program unit containing (or accessing from other modules) definitions that are to be made accessible to other
program units (14.2)

name
identifier of a program constituent, formed according to the rules given in 6.2.2

NaN
Not a Number, a symbolic floating-point datum (ISO/IEC/IEEE 60559:2011)

operand
data value that is the subject of an operator

operator
intrinsic-operator, defined-unary-op, or defined-binary-op (R608, R1003, R1023)

passed-object dummy argument
dummy argument of a type-bound procedure or procedure pointer component that becomes associated with the
object through which the procedure is invoked (7.5.4.5)

pointer
data pointer or procedure pointer

data pointer
data entity with the POINTER attribute (8.5.14)

procedure pointer
procedure with the EXTERNAL and POINTER attributes (8.5.9, 8.5.14)

local procedure pointer
procedure pointer that is part of a local variable, or a named procedure pointer that is not a dummy argument
or accessed by use or host association

pointer assignment
association of a pointer with a target, by execution of a pointer assignment statement (10.2.2) or an intrinsic
assignment statement (10.2.1.2) for a derived-type object that has the pointer as a subobject

polymorphic
⟨data entity⟩ able to be of differing dynamic types during program execution (7.3.2.3)

preconnected
⟨file or unit⟩ connected at the beginning of execution of the program (12.5.5)
3.112 procedure
entity encapsulating an arbitrary sequence of actions that can be invoked directly during program execution

3.112.1 dummy procedure
procedure that is a dummy argument (15.2.2.3)

3.112.2 external procedure
procedure defined by an external subprogram (R503) or by means other than Fortran (15.6.3)

3.112.3 internal procedure
procedure defined by an internal subprogram (R512)

3.112.4 module procedure
procedure defined by a module subprogram, or a procedure provided by an intrinsic module (R1408)

3.112.5 pure procedure
procedure declared or defined to be pure (15.7)

3.112.6 type-bound procedure
procedure that is bound to a derived type and referenced via an object of that type (7.5.5)

3.113 processor
combination of a computing system and mechanism by which programs are transformed for use on that computing system

3.114 processor dependent
not completely specified in this document, having methods and semantics determined by the processor

3.115 program
set of Fortran program units and entities defined by means other than Fortran that includes exactly one main program

3.116 program unit
main program, external subprogram, module, submodule, or block data program unit (5.2.1)

3.117 rank
number of array dimensions of a data entity (zero for a scalar entity)

3.118 record
sequence of values or characters in a file (12.2)

3.119 record file
file composed of a sequence of records (12.1)
3.120
reference
data object reference, procedure reference, or module reference

3.120.1
data object reference
appearance of a data object designator (9.1) in a context requiring its value at that point during execution

3.120.2
function reference
appearance of the procedure designator for a function, or operator symbol for a defined operation, in a context requiring execution of the function during expression evaluation (15.5.3)

3.120.3
module reference
appearance of a module name in a USE statement (14.2.2)

3.120.4
procedure reference
appearance of a procedure designator, operator symbol, or assignment symbol in a context requiring execution of the procedure at that point during execution; or occurrence of defined input/output (13.7.6) or derived-type finalization (7.5.6.2)

3.121
saved
having the SAVE attribute (8.5.16)

3.122
scalar
data entity that can be represented by a single value of the type and that is not an array (9.5)

3.123
scoping unit
BLOCK construct, derived-type definition, interface body, program unit, or subprogram, excluding all nested scoping units in it

3.123.1
block scoping unit
scoping unit of a BLOCK construct

3.124
sequence
set of elements ordered by a one-to-one correspondence with the numbers 1, 2, to n

3.125
sequence structure
scalar data object of a sequence type (7.5.2.3)

3.126
sequence type
derived type with the SEQUENCE attribute (7.5.2.3)

3.126.1
character sequence type
sequence type with no allocatable or pointer components, and whose components are all default character or of another character sequence type
3.126.2
numeric sequence type
sequence type with no allocatable or pointer components, and whose components are all default complex, default integer, default logical, default real, double precision real, or of another numeric sequence type

3.127
shape
array dimensionality of a data entity, represented as a rank-one array whose size is the rank of the data entity and whose elements are the extents of the data entity

NOTE 3.6
Thus the shape of a scalar data entity is an array with rank one and size zero.

3.128
simply contiguous
(array designator or variable) satisfying the conditions specified in 9.5.4

NOTE 3.7
These conditions are simple ones which make it clear that the designator or variable designates a contiguous array.

3.129
size
(array) total number of elements in the array

3.130
specification expression
expression satisfying the requirements specified in 10.1.11, thus being suitable for use in specifications

3.131
specific name
name that is not a generic name

3.132
standard-conforming program
program that uses only those forms and relationships described in, and has an interpretation according to, this document

3.133
statement
sequence of one or more complete or partial lines satisfying a syntax rule that ends in -stmt (6.3)

3.133.1
executable statement
end-function-stmt, end-mpprogram-stmt, end-program-stmt, end-subroutine-stmt, or statement that is a member of the syntactic class executable-construct, excluding those in the block-specification-part of a BLOCK construct

3.133.2
nonexecutable statement
statement that is not an executable statement

3.134
statement entity
entity whose identifier has the scope of a statement or part of a statement (19.1, 19.4)
3.135
statement label
label
unsigned positive number of up to five digits that refers to an individual statement (6.2.5)

3.136
storage sequence
contiguous sequence of storage units (19.5.3.2)

3.137
storage unit
character storage unit, numeric storage unit, file storage unit, or unspecified storage unit (19.5.3.2)

3.137.1
character storage unit
unit of storage that holds a default character value (19.5.3.2)

3.137.2
numeric storage unit
unit of storage that holds a default real, default integer, or default logical value (19.5.3.2)

3.137.3
unspecified storage unit
unit of storage that holds a value that is not default character, default real, double precision real, default logical,
 or default complex (19.5.3.2)

3.138
stream file
file composed of a sequence of file storage units (12.1)

3.139
structure
scalar data object of derived type (7.5)

3.139.1
structure component
component of a structure

3.139.2
structure constructor
syntax (structure-constructor, 7.5.10) that specifies a structure value or creates such a value

3.140
submodule
program unit that extends a module or another submodule (14.2.3)

3.141
subobject
portion of data object that can be referenced, and if it is a variable defined, independently of any other portion

3.142
subprogram
function-subprogram (R1529) or subroutine-subprogram (R1534)

3.142.1
external subprogram
subprogram that is not contained in a main program, module, submodule, or another subprogram
3.142.2
internal subprogram
subprogram that is contained in a main program or another subprogram

3.142.3
module subprogram
subprogram that is contained in a module or submodule but is not an internal subprogram

3.143
subroutine
procedure invoked by a CALL statement, by defined assignment, or by some operations on derived-type entities

3.143.1
atomic subroutine
intrinsic subroutine that performs an action on its ATOM argument atomically

3.143.2
collective subroutine
intrinsic subroutine that performs a calculation on a team of images without requiring synchronization

3.144
target
entity that is pointer associated with a pointer (19.5.2.2), entity on the right-hand-side of a pointer assignment statement (R1033), or entity with the TARGET attribute (8.5.17)

3.145
team
ordered set of images created by execution of a FORM TEAM statement, or the initial ordered set of all images

3.145.1
current team
team specified by the most recently executed CHANGE TEAM statement of a CHANGE TEAM construct that has not completed execution (11.1.5), or initial team if no CHANGE TEAM construct is being executed

3.145.2
initial team
team existing at the beginning of program execution, consisting of all images

3.145.3
parent team
(team except for initial) current team at time of execution of the FORM TEAM statement that created the team (11.6.9)

3.145.4
team number
(team except for initial) integer value that identifies a team within its parent team

3.146
transformational function
intrinsic function, or function in an intrinsic module, that is neither elemental nor an inquiry function

3.147
type
data type
named category of data characterized by a set of values, a syntax for denoting these values, and a set of operations that interpret and manipulate the values (7.1)
### 3.147.1 abstract type
*type with the ABSTRACT attribute (7.5.7.1)*

### 3.147.2 declared type
*type that a data entity is declared to have, either explicitly or implicitly (7.3.2, 10.1.9)*

### 3.147.3 derived type
*type defined by a type definition (7.5) or by an intrinsic module*

### 3.147.4 dynamic type
*type of a data entity at a particular point during execution of a program (7.3.2.3, 10.1.9)*

### 3.147.5 extended type
*type with the EXTENDS attribute (7.5.7.1)*

### 3.147.6 extensible type
*type that may be extended using the EXTENDS clause (7.5.7.1)*

### 3.147.7 extension type
*(of one type with respect to another) is the same type or is an extended type whose parent type is an extension type of the other type*

### 3.147.8 intrinsic type
*type defined by this document that is always accessible (7.4)*

### 3.147.9 numeric type
*one of the types integer, real, and complex*

### 3.147.10 parent type
*(extended type) type named in the EXTENDS clause*

### 3.147.11 type compatible
*compatibility of the type of one entity with respect to another for purposes such as argument association, pointer association, and allocation (7.3.2)*

### 3.147.12 type parameter
*value used to parameterize a type (7.2)*

### 3.147.12.1 assumed type parameter
*length type parameter that assumes the type parameter value from another entity*

#### NOTE 3.8

The other entity is
- the selector for an associate name,
- the constant-expr for a named constant of type character, or
NOTE 3.8 (cont.)

- the effective argument for a dummy argument.

3.147.12.2
defered type parameter
length type parameter whose value can change during execution of a program and whose type-param-value is a colon

3.147.12.3
kind type parameter
type parameter whose value is required to be defaulted or given by a constant expression

3.147.12.4
length type parameter
type parameter whose value is permitted to be assumed, deferred, or given by a specification expression

3.147.12.5
type parameter inquiry
syntax (type-param-inquiry) that is used to inquire the value of a type parameter of a data object (9.4.5)

3.147.12.6
type parameter order
ordering of the type parameters of a type (7.5.3.2) used for derived-type specifiers (derived-type-spec, 7.5.9)

3.148
ultimate argument
nondummy entity with which a dummy argument is associated via a chain of argument associations (15.5.2.3)

3.149
undefined
<data object> does not have a valid value

3.150
undefined
<pointer> does not have a pointer association status of associated or disassociated (19.5.2.2)

3.151
unit
input/output unit
means, specified by an io-unit, for referring to a file (12.5.1)

3.152
unlimited polymorphic
able to have any dynamic type during program execution (7.3.2.3)

3.153
unsaved
not having the SAVE attribute (8.5.16)

3.154
variable
data entity that can be defined and redefined during execution of a program

3.154.1
event variable
scalar variable of type EVENT_TYPE (16.10.2.10) from the intrinsic module ISO_FORTRAN_ENV
3.154.2  
local variable  
variable in a scoping unit that is not a dummy argument or part thereof, is not a global entity or part thereof, and is not an entity or part of an entity that is accessible outside that scoping unit.

3.154.3  
lock variable  
scalar variable of type LOCK_TYPE (16.10.2.19) from the intrinsic module ISO_FORTRAN_ENV.

3.154.4  
team variable  
scalar variable of type TEAM_TYPE (16.10.2.32) from the intrinsic module ISO_FORTRAN_ENV.

3.155  
vector subscript  
section-subscript that is an array (9.5.3.3.2).

3.156  
whole array  
array component or array name without further qualification (9.5.2).
4 Notation, conformance, and compatibility

4.1 Notation, symbols and abbreviated terms

4.1.1 Syntax rules

Syntax rules describe the forms that Fortran lexical tokens, statements, and constructs may take. These syntax rules are expressed in a variation of Backus-Naur form (BNF) with the following conventions.

- Characters from the Fortran character set (6.1) are interpreted literally as shown, except where otherwise noted.
- Lower-case italicized letters and words (often hyphenated and abbreviated) represent general syntactic classes for which particular syntactic entities shall be substituted in actual statements.

Common abbreviations used in syntactic terms are:

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arg</td>
<td>for argument</td>
</tr>
<tr>
<td>attr</td>
<td>for attribute</td>
</tr>
<tr>
<td>decl</td>
<td>for declaration</td>
</tr>
<tr>
<td>def</td>
<td>for definition</td>
</tr>
<tr>
<td>desc</td>
<td>for descriptor</td>
</tr>
<tr>
<td>expr</td>
<td>for expression</td>
</tr>
<tr>
<td>int</td>
<td>for integer</td>
</tr>
<tr>
<td>op</td>
<td>for operator</td>
</tr>
<tr>
<td>spec</td>
<td>for specifier</td>
</tr>
<tr>
<td>stmt</td>
<td>for statement</td>
</tr>
</tbody>
</table>

- The syntactic metasymbols used are:
  - `is` introduces a syntactic class definition
  - `or` introduces a syntactic class alternative
  - `[]` encloses an optional item
  - `[] ...` encloses an optionally repeated item that may occur zero or more times
  - `continues a syntax rule`

Each syntax rule is given a unique identifying number of the form Rsnn, where s is a one- or two-digit clause number and nn is a two-digit sequence number within that clause. The syntax rules are distributed as appropriate throughout the text, and are referenced by number as needed. Some rules in Clauses 5 and 6 are more fully described in later clauses; in such cases, the clause number s is the number of the later clause where the rule is repeated.

- The syntax rules are not a complete and accurate syntax description of Fortran, and cannot be used to generate a Fortran parser automatically; where a syntax rule is incomplete, it is restricted by corresponding constraints and text.

NOTE 4.1

An example of the use of the syntax rules is:

```
digit-string  is  digit  [  digit  ]  ...
```

The following are examples of forms for a digit string allowed by the above rule:

```
digit
digit digit
digit digit digit
digit digit digit digit
digit digit digit digit digit digit digit digit
```

If particular entities are substituted for `digit`, actual digit strings might be:
NOTE 4.1 (cont.)

| 4  |
| 67 |
| 1999 |
| 10243852 |

4.1.2 Constraints

1. Each constraint is given a unique identifying number of the form Csnn, where s is a one- or two-digit clause number and nn is a two- or three-digit sequence number within that clause.

2. Often a constraint is associated with a particular syntax rule. Where that is the case, the constraint is annotated with the syntax rule number in parentheses. A constraint that is associated with a syntax rule constitutes part of the definition of the syntax term defined by the rule. It thus applies in all places where the syntax term appears.

3. Some constraints are not associated with particular syntax rules. The effect of such a constraint is similar to that of a restriction stated in the text, except that a processor is required to have the capability to detect and report violations of constraints (4.2). In some cases, a broad requirement is stated in text and a subset of the same requirement is also stated as a constraint. This indicates that a standard-conforming program is required to adhere to the broad requirement, but that a standard-conforming processor is required only to have the capability of diagnosing violations of the constraint.

4.1.3 Assumed syntax rules

In order to minimize the number of additional syntax rules and convey appropriate constraint information, the following rules are assumed.

R401 $xyz$-list is $xyz$, $xyz$ [...]

R402 $xyz$-name is name

R403 scalar-$xyz$ is $xyz$

C401 (R403) scalar-$xyz$ shall be scalar.

2. The letters “$xyz$” stand for any syntactic class phrase. An explicit syntax rule for a term overrides an assumed rule.

4.1.4 Syntax conventions and characteristics

1. Any syntactic class name ending in “$-stmt$” follows the source form statement rules: it shall be delimited by end-of-line or semicolon, and may be labeled unless it forms part of another statement (such as an IF or WHERE statement). Conversely, everything considered to be a source form statement is given a “$-stmt$” ending in the syntax rules.

2. The rules on statement ordering are described rigorously in the definition of program-unit (R502). Expression hierarchy is described rigorously in the definition of expr (R1022).

3. The suffix “$-spec$” is used consistently for specifiers, such as input/output statement specifiers. It also is used for type declaration attribute specifications (for example, “$array-spec$” in R815), and in a few other cases.

4. Where reference is made to a type parameter, including the surrounding parentheses, the suffix “$-selector$” is used. See, for example, “$kind-selector$” (R706) and “length-selector” (R722).
4.1.5 Text conventions

In descriptive text, an equivalent English word is frequently used in place of a syntactic term. Particular statements and attributes are identified in the text by an upper-case keyword, e.g., “END statement”. The descriptions of obsolescent features appear in a smaller type size.

NOTE 4.2
This sentence is an example of the type size used for obsolescent features.

4.2 Conformance

A program (5.2.2) is a standard-conforming program if it uses only those forms and relationships described herein and if the program has an interpretation according to this document. A program unit (5.2.1) conforms to this document if it can be included in a program in a manner that allows the program to be standard conforming.

A processor conforms to this document if:

1. it executes any standard-conforming program in a manner that fulfills the interpretations herein, subject to any limits that the processor may impose on the size and complexity of the program;
2. it contains the capability to detect and report the use within a submitted program unit of a form designated herein as obsolescent, insofar as such use can be detected by reference to the numbered syntax rules and constraints;
3. it contains the capability to detect and report the use within a submitted program unit of a form or relationship that is not permitted by the numbered syntax rules or constraints, including the deleted features described in Annex B;
4. it contains the capability to detect and report the use within a submitted program unit of an intrinsic type with a kind type parameter value not supported by the processor (7.4);
5. it contains the capability to detect and report the use within a submitted program unit of source form or characters not permitted by Clause 6;
6. it contains the capability to detect and report the use within a submitted program of name usage not consistent with the scope rules for names, labels, operators, and assignment symbols in Clause 19;
7. it contains the capability to detect and report the use within a submitted program unit of a non-standard intrinsic procedure (including one with the same name as a standard intrinsic procedure but with different requirements);
8. it contains the capability to detect and report the use within a submitted program unit of a non-standard intrinsic module;
9. it contains the capability to detect and report the use within a submitted program unit of a procedure from a standard intrinsic module, if the procedure is not defined by this document or the procedure has different requirements from those specified by this document; and
10. it contains the capability to detect and report the reason for rejecting a submitted program.

However, in a format specification that is not part of a FORMAT statement (13.2.1), a processor need not detect or report the use of deleted or obsolescent features, or the use of additional forms or relationships.

A standard-conforming processor may allow additional forms and relationships provided that such additions do not conflict with the standard forms and relationships. However, a standard-conforming processor may allow additional intrinsic procedures even though this could cause a conflict with the name of a procedure in a standard-conforming program. If such a conflict occurs and involves the name of an external procedure, the processor is permitted to use the intrinsic procedure unless the name has the EXTERNAL attribute (8.5.9) where it is used. A standard-conforming program shall not use nonstandard intrinsic procedures or modules that have been added by the processor.

Because a standard-conforming program may place demands on a processor that are not within the scope of this document or may include standard items that are not portable, such as external procedures defined by means...
other than Fortran, conformance to this document does not ensure that a program will execute consistently on all or any standard-conforming processors.

The semantics of facilities that are identified as processor dependent are not completely specified in this document. They shall be provided, with methods or semantics determined by the processor.

The processor should be accompanied by documentation that specifies the limits it imposes on the size and complexity of a program and the means of reporting when these limits are exceeded, that defines the additional forms and relationships it allows, and that defines the means of reporting the use of additional forms and relationships and the use of deleted or obsolescent forms. In this context, the use of a deleted form is the use of an additional form.

The processor should be accompanied by documentation that specifies the methods or semantics of processor-dependent facilities.

### 4.3 Compatibility

#### 4.3.1 Previous Fortran standards

Table 4.3 lists the previous editions of the Fortran International Standard, along with their informal names.

<table>
<thead>
<tr>
<th>Official designation</th>
<th>Informal name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISO R 1539-1972</td>
<td>Fortran 66</td>
</tr>
<tr>
<td>ISO 1539-1980</td>
<td>Fortran 77</td>
</tr>
<tr>
<td>ISO/IEC 1539:1991</td>
<td>Fortran 90</td>
</tr>
<tr>
<td>ISO/IEC 1539-1:1997</td>
<td>Fortran 95</td>
</tr>
</tbody>
</table>

#### 4.3.2 New intrinsic procedures

Each Fortran International Standard since ISO 1539:1980 (Fortran 77), defines more intrinsic procedures than the previous one. Therefore, a Fortran program conforming to an older standard might have a different interpretation under a newer standard if it invokes an external procedure having the same name as one of the new standard intrinsic procedures, unless that procedure is specified to have the EXTERNAL attribute.

#### 4.3.3 Fortran 2008 compatibility

Except as identified in this subclause, this document is an upward compatible extension to the preceding Fortran International Standard, ISO/IEC 1539-1:2010 (Fortran 2008). Except for the deleted features noted in Annex B.2, any standard-conforming Fortran 2008 program remains standard-conforming under this document.

Fortran 2008 specifies that the IOSTAT= variable shall be set to a processor-dependent negative value if the flush operation is not supported for the unit specified. This document specifies that the processor-dependent negative integer value shall be different from the named constants IOSTAT_EOR or IOSTAT_END from the intrinsic module ISO_FORTRAN_ENV.

Fortran 2008 permitted a noncontiguous array that was supplied as an actual argument corresponding to a contiguous INTENT (INOUT) dummy argument in one iteration of a DO CONCURRENT construct, without being previously defined in that iteration, to be defined in another iteration; this document does not permit this.

Fortran 2008 permitted a pure statement function to reference a volatile variable, and permitted a local variable of a pure subprogram or of a BLOCK construct construct within a pure subprogram to be volatile (provided it...
was not used); this document does not permit that.

### 4.3.4 Fortran 2003 compatibility

1. Except as identified in this subclause, this document is an upward compatible extension to ISO/IEC 1539-1:2004 (Fortran 2003). Except as identified in this subclause, any standard-conforming Fortran 2003 program remains standard-conforming under this document.
2. Fortran 2003 permitted a sequence type to have type parameters; that is not permitted by this document.
3. Fortran 2003 specified that array constructors and structure constructors of finalizable type are finalized. This document specifies that these constructors are not finalized.
4. The form produced by the G edit descriptor for some values and some input/output rounding modes differs from that specified by Fortran 2003.
5. Fortran 2003 permitted a derived type to have type parameters; that is not permitted by this document.
6. Fortran 2003 specified that array constructors and structure constructors of finalizable type are finalized. This document specifies that these constructors are not finalized.
7. The form produced by the G edit descriptor for some values and some input/output rounding modes differs from that specified by Fortran 2003.

### 4.3.5 Fortran 95 compatibility

1. Except as identified in this subclause, this document is an upward compatible extension to ISO/IEC 1539-1:1997 (Fortran 95). Except as identified in this subclause, any standard-conforming Fortran 95 program remains standard-conforming under this document.
2. Fortran 95 permitted defined assignment between character strings of the same rank and different kinds. This document does not permit that if both of the different kinds are ASCII, ISO 10646, or default kind.
3. The following Fortran 95 features might have different interpretations in this document.
   - Earlier Fortran standards had the concept of printing, meaning that column one of formatted output had special meaning for a processor-dependent (possibly empty) set of external files. This could be neither detected nor specified by a standard-specified means. The interpretation of the first column is not specified by this document.
   - This document specifies a different output format for real zero values in list-directed and namelist output.
   - If the processor distinguishes between positive and negative real zero, this document requires different returned values for ATAN2(Y,X) when X < 0 and Y is negative real zero and for LOG(X) and SQRT(X) when X is complex with X%RE < 0 and X%IM is negative real zero.
   - This document has fewer restrictions on constant expressions than Fortran 95; this might affect whether a variable is considered to be an automatic data object.
The form produced by the G edit descriptor with \( d \) equal to zero differs from that specified by Fortran 95 for some values.

### 4.3.6 Fortran 90 compatibility

Except for the deleted features noted in Annex B.1, and except as identified in this subclause, this document is an upward compatible extension to ISO/IEC 1539:1991 (Fortran 90). Any standard-conforming Fortran 90 program that does not use one of the deleted features remains standard-conforming under this document.

2. The PAD= specifier in the INQUIRE statement in this document returns the value UNDEFINED if there is no connection or the connection is for unformatted input/output. Fortran 90 specified YES.

3. Fortran 90 specified that if the second argument to MOD or MODULO was zero, the result was processor dependent. This document specifies that the second argument shall not be zero.

4. Fortran 90 permitted defined assignment between character strings of the same rank and different kinds. This document does not permit that if both of the different kinds are ASCII, ISO 10646, or default kind.

5. The following Fortran 90 features have different interpretations in this document:
   - if the processor distinguishes between positive and negative real zero, the result value of the intrinsic function SIGN when the second argument is a negative real zero;
   - formatted output of negative real values (when the output value is zero);
   - whether an expression is a constant expression (thus whether a variable is considered to be an automatic data object);
   - the G edit descriptor with \( d \) equal to zero for some values.

### 4.3.7 FORTRAN 77 compatibility

Except for the deleted features noted in Annex B.1, and except as identified in this subclause, this document is an upward compatible extension to ISO 1539:1980 (FORTRAN 77). Any standard-conforming FORTRAN 77 program that does not use one of the deleted features noted in Annex B.1 and that does not depend on the differences specified here remains standard-conforming under this document. This document restricts the behavior for some features that were processor dependent in FORTRAN 77. Therefore, a standard-conforming FORTRAN 77 program that uses one of these processor-dependent features might have a different interpretation under this document, yet remain a standard-conforming program. The following FORTRAN 77 features might have different interpretations in this document.

- FORTRAN 77 permitted a processor to supply more precision derived from a default real constant than can be represented in a default real datum when the constant is used to initialize a double precision real data object in a DATA statement. This document does not permit a processor this option.
- If a named variable that was not in a common block was initialized in a DATA statement and did not have the SAVE attribute specified, FORTRAN 77 left its SAVE attribute processor dependent. This document specifies (8.6.7) that this named variable has the SAVE attribute.
- FORTRAN 77 specified that the number of characters required by the input list was to be less than or equal to the number of characters in the record during formatted input. This document specifies (12.6.4.5.3) that the input record is logically padded with blanks if there are not enough characters in the record, unless the PAD= specifier with the value ‘NO’ is specified in an appropriate OPEN or READ statement.
- A value of 0 for a list item in a formatted output statement will be formatted in a different form for some G edit descriptors. In addition, this document specifies how rounding of values will affect the output field form, but FORTRAN 77 did not address this issue. Therefore, the form produced for certain combinations of values and G edit descriptors might differ from that produced by some FORTRAN 77 processors.
- FORTRAN 77 did not permit a processor to distinguish between positive and negative real zero; if the processor does so distinguish, the result will differ for the intrinsic function SIGN when the second argument is negative real zero, and formatted output of negative real zero will be different.
4.4 Deleted and obsolescent features

4.4.1 General

This document protects the users’ investment in existing software by including all but six of the language elements of Fortran 90 that are not processor dependent. This document identifies two categories of outmoded features. The first category, deleted features, consists of features considered to have been redundant in FORTRAN 77 and largely unused in Fortran 90. Those in the second category, obsolescent features, are considered to have been redundant in Fortran 90 and Fortran 95, but are still frequently used.

4.4.2 Nature of deleted features

Better methods existed in FORTRAN 77 for each deleted feature. These features were not included in Fortran 95 or Fortran 2003, and are not included in this revision of Fortran.

4.4.3 Nature of obsolescent features

Better methods existed in Fortran 90 and Fortran 95 for each obsolescent feature. It is recommended that programmers use these better methods in new programs and convert existing code to these methods.

The obsolescent features are identified in the text of this document by a distinguishing type font (4.1.5).

A future revision of this document might delete an obsolescent feature if its use has become insignificant.
5  Fortran concepts

5.1 High level syntax

This subclause introduces the syntax associated with program units and other Fortran concepts above the construct, statement, and expression levels and illustrates their relationships.

NOTE 5.1

Constraints and other information related to the rules that do not begin with R5 appear in the appropriate clause.

R501  program  is  program-unit

R502  program-unit  is  main-program
or  external-subprogram
or  module
or  submodule
or  block-data

R1401  main-program  is  [ program-stmt ]

R503  external-subprogram  is  function-subprogram
or  subroutine-subprogram

R1529  function-subprogram  is  function-stmt

R1534  subroutine-subprogram  is  subroutine-stmt

R1404  module  is  module-stmt

R1416  submodule  is  submodule-stmt

R1420  block-data  is  block-data-stmt
R504 specification-part is [ use-stmt ] ...
[ import-stmt ] ...
[ implicit-part ] ...
[ declaration-construct ] ...

R505 implicit-part is [ implicit-part-stmt ] ...

R506 implicit-part-stmt is implicit-stmt
or parameter-stmt
or format-stmt
or entry-stmt

R507 declaration-construct is specification-construct
or data-stmt
or format-stmt
or entry-stmt
or stmt-function-stmt

R508 specification-construct is derived-type-def
or enum-def
or generic-stmt
or interface-block
or parameter-stmt
or procedure-declaration-stmt
or other-specification-stmt
or type-declaration-stmt

R509 execution-part is executable-construct
[ execution-part-construct ] ...

R510 execution-part-construct is executable-construct
or format-stmt
or entry-stmt
or data-stmt

R511 internal-subprogram-part is contains-stmt
[ internal-subprogram ] ...

R512 internal-subprogram is function-subprogram
or subroutine-subprogram

R1407 module-subprogram-part is contains-stmt
[ module-subprogram ] ...

R1408 module-subprogram is function-subprogram
or subroutine-subprogram
or separate-module-subprogram

R1538 separate-module-subprogram is mp-subprogram-stmt
[ specification-part ]
[ execution-part ]
[ internal-subprogram-part ]
end-mp-subprogram-stmt

R513 other-specification-stmt is access-stmt
or allocatable-stmt

36 ISO/IEC JTC 1/SC 22/WG5/N2118
or asynchronous-stmt
or bind-stmt
or codimension-stmt
or contiguous-stmt
or dimension-stmt
or external-stmt
or intent-stmt
or intrinsic-stmt
or namelist-stmt
or optional-stmt
or pointer-stmt
or protected-stmt
or save-stmt
or target-stmt
or volatile-stmt
or value-stmt
or common-stmt
or equivalence-stmt

R514 executable-construct is action-stmt
or associate-construct
or block-construct
or case-construct
or change-team-construct
or critical-construct
or do-construct
or if-construct
or select-rank-construct
or select-type-construct
or where-construct
or forall-construct

R515 action-stmt is allocate-stmt
or assignment-stmt
or backspace-stmt
or call-stmt
or close-stmt
or continue-stmt
or cycle-stmt
or deallocate-stmt
or endfile-stmt
or error-stop-stmt
or event-post-stmt
or event-wait-stmt
or exit-stmt
or fail-image-stmt
or flush-stmt
or form-team-stmt
or goto-stmt
or if-stmt
or inquire-stmt
or lock-stmt
or nullify-stmt
or open-stmt
or pointer-assignment-stmt
or print-stmt
5.2 Program unit concepts

5.2.1 Program units and scoping units

1 Program units are the fundamental components of a Fortran program. A program unit is a main program, an external subprogram, a module, a submodule, or a block data program unit.

2 A subprogram is a function subprogram or a subroutine subprogram. A module contains definitions that can be made accessible to other program units. A submodule is an extension of a module; it may contain the definitions of procedures declared in a module or another submodule. A block data program unit is used to specify initial values for data objects in named common blocks.

3 Each type of program unit is described in Clause 14 or 15.

4 A program unit consists of a set of nonoverlapping scoping units.

NOTE 5.2 The module or submodule containing a module subprogram is the host scoping unit of the module subprogram. The containing main program or subprogram is the host scoping unit of an internal subprogram. An internal procedure is local to its host in the sense that its name is accessible within the host scoping unit and all its other internal procedures but is not accessible elsewhere.

5.2.2 Program

1 A program shall consist of exactly one main program, any number (including zero) of other kinds of program units, any number (including zero) of external procedures, and any number (including zero) of other entities defined by means other than Fortran. The main program shall be defined by a Fortran main-program program-unit or by means other than Fortran, but not both.

5.2.3 Procedure

1 A procedure is either a function or a subroutine. Invocation of a function in an expression causes a value to be computed which is then used in evaluating the expression.

2 A procedure that is not pure might change the program state by changing the value of accessible data objects or procedure pointers.

3 Procedures are described further in Clause 15.
5.2.4 Module

1 A module contains (or accesses from other modules) definitions that can be made accessible to other program units. These definitions include data object declarations, type definitions, procedure definitions, and interface blocks. Modules are further described in Clause 14.

5.2.5 Submodule

1 A submodule extends a module or another submodule.

2 It may provide definitions (15.6) for procedures whose interfaces are declared (15.4.3.2) in an ancestor module or submodule. It may also contain declarations and definitions of other entities, which are accessible in its descendants. An entity declared in a submodule is not accessible by use association unless it is a module procedure whose interface is declared in the ancestor module. Submodules are further described in Clause 14.

NOTE 5.3
A submodule has access to entities in its parent module or submodule by host association.

5.3 Execution concepts

5.3.1 Statement classification

1 Each Fortran statement is classified as either an executable statement or a nonexecutable statement.

2 An executable statement is an instruction to perform or control an action. Thus, the executable statements of a program unit determine the behavior of the program unit.

3 Nonexecutable statements are used to configure the program environment in which actions take place.

5.3.2 Statement order

Table 5.1: Requirements on statement ordering

<table>
<thead>
<tr>
<th>PROGRAM, FUNCTION, SUBROUTINE, MODULE, SUBMODULE, or BLOCK DATA statement</th>
<th>USE statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPORT statements</td>
<td></td>
</tr>
<tr>
<td>FORMAT and ENTRY statements</td>
<td></td>
</tr>
<tr>
<td>DATA statements</td>
<td></td>
</tr>
<tr>
<td>CONTAINS statement</td>
<td></td>
</tr>
<tr>
<td>Internal subprograms or module subprograms</td>
<td></td>
</tr>
<tr>
<td>END statement</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PARAMETER statements</th>
<th>IMPLICIT statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Derived-type definitions, interface blocks, type declaration statements, enumeration definitions, procedure declarations, specification statements, and statement function statements</td>
<td>Derived-type definitions, interface blocks, type declaration statements, enumeration definitions, procedure declarations, specification statements, and statement function statements</td>
</tr>
</tbody>
</table>

PARAMETER and DATA statements

<table>
<thead>
<tr>
<th>IMPLICIT NONE</th>
</tr>
</thead>
</table>

IMPLICIT statement
The syntax rules of subclause 5.1 specify the statement order within program units and subprograms. These rules are illustrated in Table 5.1 and Table 5.2. Table 5.1 shows the ordering rules for statements and applies to all program units, subprograms, and interface bodies. Vertical lines delineate varieties of statements that may be interspersed and horizontal lines delineate varieties of statements that shall not be interspersed. Internal or module subprograms shall follow a CONTAINS statement. Between USE and CONTAINS statements in a subprogram, nonexecutable statements generally precede executable statements, although the ENTRY statement, FORMAT statement, and DATA statement may appear among the executable statements. Table 5.2 shows which statements are allowed in some kinds of scoping units.

<table>
<thead>
<tr>
<th>Statement type</th>
<th>Kind of scoping unit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Main program</td>
</tr>
<tr>
<td>USE</td>
<td>Yes</td>
</tr>
<tr>
<td>IMPORT</td>
<td>No</td>
</tr>
<tr>
<td>ENTRY</td>
<td>No</td>
</tr>
<tr>
<td>FORMAT</td>
<td>Yes</td>
</tr>
<tr>
<td>Misc. decls</td>
<td>Yes</td>
</tr>
<tr>
<td>DATA</td>
<td>Yes</td>
</tr>
<tr>
<td>Derived-type</td>
<td>Yes</td>
</tr>
<tr>
<td>Interface</td>
<td>Yes</td>
</tr>
<tr>
<td>Executable</td>
<td>Yes</td>
</tr>
<tr>
<td>CONTAINS</td>
<td>Yes</td>
</tr>
</tbody>
</table>

(1) Miscellaneous declarations are PARAMETER statements, IMPLICIT statements, type declaration statements, enumeration definitions, procedure declaration statements, and specification statements.

5.3.3 The END statement

Each program unit, module subprogram, and internal subprogram shall have exactly one END statement. The end-program-stmt, end-function-stmt, end-subroutine-stmt, and end-mp-subprogram-stmt statements are executable, and may be branch target statements (11.2). Executing an end-program-stmt initiates normal termination. Executing an end-function-stmt, end-subroutine-stmt, or end-mp-subprogram-stmt is equivalent to executing a return-stmt with no scalar-int-expr.

The end-module-stmt, end-submodule-stmt, and end-block-data-stmt statements are nonexecutable.

5.3.4 Program execution

1 Execuption of a program consists of the asynchronous execution of a fixed number (which may be one) of its images. Each image has its own execution state, floating-point status (17.7), and set of data objects, input/output units, and procedure pointers. The image index that identifies an image is an integer value in the range one to the number of images in a team.

2 A team is an ordered set of images that is either the initial team, consisting of all images, or a subset of a parent team formed by execution of a FORM TEAM statement. The initial team has no parent; every other team has a unique parent team. Within its parent team, each team is identified by its team number; this is the integer value that was specified in the FORM TEAM statement.

3 During execution, each image has a current team, which is only changed by execution of CHANGE TEAM and END TEAM statements. Image indices, and thus coindexing of variable names with an image-selector, are
relative to the current team unless a different team is specified. Initially, the current team is the initial team.

NOTE 5.4

Fortran control constructs (11.1, 11.2) control the progress of execution in each image. Image control statements (11.6.1) affect the relative progress of execution between images. Coarrays (5.4.7) provide a mechanism for accessing data on one image from another image.

NOTE 5.5

A processor might allow the number of images to be chosen at compile time, link time, or run time. It might be the same as the number of CPUs but this is not required. Compiling for a single image might permit the optimizer to eliminate overhead associated with parallel execution. A program that makes assumptions about the number of images is unlikely to be portable.

5.3.5 Execution sequence

Following the creation of a fixed number of images, execution begins on each image. Image execution is a sequence, in time, of actions. Actions take place during execution of the statement that performs them (except when explicitly stated otherwise). Segments (11.6.2) executed by a single image are totally ordered, and segments executed by separate images are partially ordered by image control statements (11.6.1).

If the program contains a Fortran main program, each image begins execution with the first executable construct of the main program. The execution of a main program or subprogram involves execution of the executable constructs within its scoping unit. When a Fortran procedure is invoked, the specification expressions within the specification-part of the invoked procedure, if any, are evaluated in a processor dependent order. Thereafter, execution proceeds to the first executable construct appearing within the scoping unit of the procedure after the invoked entry point. With the following exceptions, the effect of execution is as if the executable constructs are executed in the order in which they appear in the main program or subprogram until a STOP, ERROR STOP, RETURN, or END statement is executed.

- Execution of a branching statement (11.2) changes the execution sequence. These statements explicitly specify a new starting place for the execution sequence.
- DO constructs, IF constructs, SELECT CASE constructs, SELECT RANK constructs, and SELECT TYPE constructs contain an internal statement structure and execution of these constructs involves implicit internal transfer of control. See Clause 11 for the detailed semantics of each of these constructs.
- A BLOCK construct may contain specification expressions; see 11.1.4 for detailed semantics of this construct.
- An END=, ERR=, or EOR= specifier (12.11) might result in a branch.
- An alternate return might result in a branch.

5.3.6 Image execution states

There are three image execution states: active, stopped, and failed. An image that has initiated normal termination of execution is a stopped image. An image that has ceased participating in program execution but has not initiated termination is a failed image. All other images are active.

A failed image remains failed for the remainder of the execution of the program. The conditions that cause an image to fail are processor dependent.

Defining a coindexed object on a failed image has no effect other than defining the stat-variable, if one appears, with the value STAT_FAILED_IMAGE (16.10.2.26). The value of a reference to a coindexed object on a failed image is processor dependent. Execution continues after such a reference.

When an image fails during the execution of a segment, a data object on a nonfailed image becomes undefined if it is not a lock variable or event variable and it might be defined or become undefined by execution of a statement of the segment other than an invocation of an atomic subroutine with the object as an actual argument corresponding to the ATOM dummy argument.
5.3.7 Termination of execution

Termination of execution of a program is either normal termination or error termination. Normal termination occurs only when all images initiate normal termination and occurs in three steps: initiation, synchronization, and completion. In this case, all images synchronize execution at the second step so that no image starts the completion step until all images have finished the initiation step. Error termination occurs when any image initiates error termination. Once error termination has been initiated on an image, error termination is initiated on all images that have not already initiated error termination. Termination of execution of the program occurs when all images have terminated execution.

Normal termination of execution of an image is initiated when a STOP statement or end-program-stmt is executed. Normal termination of execution of an image also may be initiated during execution of a procedure defined by a companion processor (ISO/IEC 9899:2011 5.1.2.2.3 and 7.22.4.4). If normal termination of execution is initiated within a Fortran program unit and the program incorporates procedures defined by a companion processor, the process of execution termination shall include the effect of executing the C exit() function (ISO/IEC 9899:2011 7.22.4.4) during the completion step.

Error termination of execution of an image is initiated if an ERROR STOP statement is executed or as specified elsewhere in this document. When error termination on an image has been initiated, the processor should initiate error termination on other images as quickly as possible.

If the processor supports the concept of a process exit status, it is recommended that error termination initiated other than by an ERROR STOP statement supplies a processor-dependent nonzero value as the process exit status.

NOTE 5.6
As well as in the circumstances specified in this document, error termination might be initiated by means other than Fortran.

NOTE 5.7
If an image has initiated normal termination, its data remain available for possible reference or definition by other images that are still executing.

5.4 Data concepts

5.4.1 Type

A type is a named categorization of data that, together with its type parameters, determines the set of values, syntax for denoting these values, and the set of operations that interpret and manipulate the values. This central concept is described in 7.1.

A type is either an intrinsic type or a derived type.

5.4.1.1 Intrinsic type

The intrinsic types are integer, real, complex, character, and logical. The properties of intrinsic types are described in 7.4.

All intrinsic types have a kind type parameter called KIND, which determines the representation method for the specified type. The intrinsic type character also has a length type parameter called LEN, which determines the length of the character string.

5.4.1.2 Derived type

Derived types may be parameterized. A scalar object of derived type is a structure; assignment of structures is defined intrinsically (10.2.1.3), but there are no intrinsic operations for structures. For each derived type, a
structure constructor is available to create values (7.5.10). In addition, objects of derived type may be used as procedure arguments and function results, and may appear in input/output lists. If additional operations are needed for a derived type, they shall be defined by procedures (10.1.6).

2 Derived types are described further in 7.5.

5.4.2 Data value

1 Each intrinsic type has associated with it a set of values that a datum of that type may take, depending on the values of the type parameters. The values for each intrinsic type are described in 7.4. The values that objects of a derived type may assume are determined by the type definition, type parameter values, and the sets of values of its components.

5.4.3 Data entity

5.4.3.1 General

1 A data entity has a type and type parameters; it might have a data value (an exception is an undefined variable). Every data entity has a rank and is thus either a scalar or an array.

2 A data entity that is the result of the execution of a function reference is called the function result.

5.4.3.2 Data object

1 A data object is either a constant, variable, or a subobject of a constant. The type and type parameters of a named data object may be specified explicitly (8.2) or implicitly (8.7).

2 Subobjects are portions of data objects that may be referenced and defined (variables only) independently of the other portions.

3 These include portions of arrays (array elements and array sections), portions of character strings (substrings), portions of complex objects (real and imaginary parts), and portions of structures (components). Subobjects are themselves data objects, but subobjects are referenced only by object designators or intrinsic functions. A subobject of a variable is a variable. Subobjects are described in Clause 9.

4 The following objects are referenced by a name:
   • a named scalar (a scalar object);
   • a named array (an array object).

5 The following subobjects are referenced by an object designator:
   • an array element (a scalar subobject);
   • an array section (an array subobject);
   • a complex part designator (the real or imaginary part of a complex object);
   • a structure component (a scalar or an array subobject);
   • a substring (a scalar subobject).

5.4.3.2.1 Variable

1 A variable can have a value or be undefined; during execution of a program it can be defined, redefined, or become undefined.

2 A local variable of a module, submodule, main program, subprogram, or BLOCK construct is accessible only in that scoping unit or construct and in any contained scoping units and constructs.

NOTE 5.8
A subobject of a local variable is also a local variable.
NOTE 5.8 (cont.)

A local variable cannot be in COMMON or have the BIND attribute, because common blocks and variables with the BIND attribute are global entities.

5.4.3.2.2 Constant

1 A constant is either a named constant or a literal constant.

2 Named constants are defined using the PARAMETER attribute (8.5.13, 8.6.11). The syntax of literal constants is described in 7.4.

5.4.3.2.3 Subobject of a constant

1 A subobject of a constant is a portion of a constant.

2 In an object designator for a subobject of a constant, the portion referenced may depend on the value of a variable.

NOTE 5.9

For example, given:

```plaintext
CHARACTER (LEN = 10), PARAMETER :: DIGITS = '0123456789'
CHARACTER (LEN = 1) :: DIGIT
INTEGER :: I

... DIGIT = DIGITS (I:1)
```

DIGITS is a named constant and DIGITS (I:1) designates a subobject of the constant DIGITS.

5.4.3.3 Expression

1 An expression (10.1) produces a data entity when evaluated. An expression represents either a data object reference or a computation; it is formed from operands, operators, and parentheses. The type, type parameters, value, and rank of an expression result are determined by the rules in Clause 10.

5.4.3.4 Function reference

1 A function reference produces a data entity when the function is executed during expression evaluation. The type, type parameters, and rank of a function result are determined by the interface of the function (15.3.3). The value of a function result is determined by execution of the function.

5.4.4 Definition of objects and pointers

1 When an object is given a valid value during program execution, it becomes defined. This is often accomplished by execution of an assignment or input statement. When a variable does not have a predictable value, it is undefined.

2 Similarly, when a pointer is associated with a target or nullified, its pointer association status becomes defined. When the association status of a pointer is not predictable, its pointer association status is undefined.

3 Clause 19 describes the ways in which variables become defined and undefined and the association status of pointers becomes defined and undefined.

5.4.5 Reference

1 A data object is referenced when its value is required during execution. A procedure is referenced when it is executed.
The appearance of a data object designator or procedure designator as an actual argument does not constitute a reference to that data object or procedure unless such a reference is necessary to complete the specification of the actual argument.

5.4.6 Array

An array may have up to fifteen dimensions minus its corank, and any extent in any dimension. The size of an array is the total number of elements, which is equal to the product of the extents. An array may have zero size. The shape of an array is determined by its rank and its extent in each dimension, and is represented as a rank-one array whose elements are the extents. All named arrays shall be declared, and the rank of a named array is specified in its declaration. Except for an assumed-rank array, the rank of a named array, once declared, is constant; the extents may be constant or may vary during execution.

Any intrinsic operation defined for scalar objects may be applied to conformable objects. Such operations are performed elementally to produce a resultant array conformable with the array operands. If an elemental operation is intrinsically pure or is implemented by a pure elemental function (15.8), the element operations may be performed simultaneously or in any order.

A rank-one array may be constructed from scalars and other arrays and may be reshaped into any allowable array shape (7.8).

Arrays may be of any type and are described further in 9.5.

5.4.7 Coarray

A coarray is a data entity that has nonzero corank; it can be directly referenced or defined by other images. It may be a scalar or an array.

For each coarray on an image, there is a corresponding coarray with the same type, type parameters, and bounds on every other image that is in the same team.

The set of corresponding coarrays on all images in a team is arranged in a rectangular pattern. The dimensions of this pattern are the codimensions; the number of codimensions is the corank. The bounds for each codimension are the cobounds.

NOTE 5.10
If the total number of images is not a multiple of the product of the sizes of each but the rightmost of the codimensions, the rectangular pattern will be incomplete.

A coarray on any image can be accessed directly by using cosubscripts. On its own image, a coarray can also be accessed without use of cosubscripts.

A subobject of a coarray is a coarray if it does not have any cosubscripts, vector subscripts, allocatable component selection, or pointer component selection.

For a coindexed object, its cosubscript list determines the image index (9.6) in the same way that a subscript list determines the subscript order value for an array element (9.5.3.2).

Intrinsic procedures are provided for mapping between an image index and a list of cosubscripts.

NOTE 5.11
The mechanism for an image to reference and define a coarray on another image might vary according to the hardware. On a shared-memory machine, a coarray on an image and the corresponding coarrays on other images could be implemented as a sequence of arrays with evenly spaced starting addresses. On a distributed-memory machine with separate physical memory for each image, a processor might store a coarray at the same virtual address in each physical memory.
NOTE 5.12

Except in contexts where coindexed objects are disallowed, accessing a coarray on its own image by using a set of cosubscripts that specify that image has the same effect as accessing it without cosubscripts. In particular, the segment ordering rules (11.6.2) apply whether or not cosubscripts are used to access the coarray.

5.4.8 Established coarrays

1. A nonallocatable coarray with the SAVE attribute is established in the initial team.
2. An allocated allocatable coarray is established in the team in which it was allocated. An unallocated allocatable coarray is not established.
3. A coarray that is an associating entity in a coarray-association of a CHANGE TEAM statement is established in the team of its CHANGE TEAM construct.
4. A nonallocatable coarray that is a dummy argument or host associated with a dummy argument is established in the team in which the procedure was invoked. A coarray dummy argument is not established in any ancestor team even if the corresponding actual argument is established in one or more of them.
5. A nonallocatable coarray that is a local variable of a subprogram or host associated with a local variable of a subprogram is established in the team in which the procedure was invoked. A nonallocatable coarray declared in a BLOCK construct is established in the team in which the BLOCK statement was executed.

5.4.9 Pointer

1. A pointer has an association status which is either associated, disassociated, or undefined (19.5.2.2).
2. A pointer that is not associated shall not be referenced or defined.
3. If a data pointer is an array, the rank is declared, but the bounds are determined when it is associated with a target.

5.4.10 Allocatable variables

1. The allocation status of an allocatable variable is either allocated or unallocated. An allocatable variable becomes allocated as described in 9.7.1.3. It becomes unallocated as described in 9.7.3.2.
2. An unallocated allocatable variable shall not be referenced or defined.
3. If an allocatable variable is an array, the rank is declared, but the bounds are determined when it is allocated. If an allocatable variable is a coarray, the corank is declared, but the cobounds are determined when it is allocated.

5.4.11 Storage

1. Many of the facilities of this document make no assumptions about the physical storage characteristics of data objects. However, program units that include storage association dependent features shall observe the storage restrictions described in 19.5.3.

5.5 Fundamental concepts

5.5.1 Names and designators

1. A name is used to identify a program constituent, such as a program unit, named variable, named constant, dummy argument, or derived type.
A designator is used to identify a program constituent or a part thereof.

### 5.5.2 Statement keyword

A statement keyword is not a reserved word; that is, a name with the same spelling is allowed. In the syntax rules, such keywords appear literally. In descriptive text, this meaning is denoted by the term “keyword” without any modifier. Examples of statement keywords are IF, READ, UNIT, KIND, and INTEGER.

### 5.5.3 Other keywords

Other keywords denote names that identify items in a list. In this case, items are identified by a preceding keyword= rather than their position within the list.

An argument keyword is the name of a dummy argument in the interface for the procedure being referenced, and may appear in an actual argument list. A type parameter keyword is the name of a type parameter in the type being specified, and may appear in a type parameter list. A component keyword is the name of a component in a structure constructor.

R516 keyword is name

### NOTE 5.13

Use of keywords rather than position to identify items in a list can make such lists more readable and allows them to be reordered. This facilitates specification of a list in cases where optional items are omitted.

### 5.5.4 Association

Name association (19.5.1) permits an entity to be identified by different names in the same scoping unit or by the same name or different names in different scoping units.

Pointer association (19.5.2) between a pointer and a target allows the target to be denoted by the pointer.

Storage association (19.5.3) causes different entities to use the same storage.

Inheritance association (19.5.4) occurs between components of the parent component and components inherited by type extension.

### 5.5.5 Intrinsic

All intrinsic types, procedures, assignments, and operators may be used in any scoping unit without further definition or specification. Intrinsic modules (16.10, 17, 18.2) may be accessed by use association.

### 5.5.6 Operator

This document specifies a number of intrinsic operators (e.g., the arithmetic operators +, -, *, /, and ** with numeric operands and the logical operators .AND., .OR., etc. with logical operands). Additional operators may be defined within a program (7.5.5, 15.4.3.4).

### 5.5.7 Companion processors

A processor has one or more companion processors. A companion processor may be a mechanism that references and defines such entities by a means other than Fortran (15.6.3), it may be the Fortran processor itself, or it may be another Fortran processor. If there is more than one companion processor, the means by which the Fortran processor selects among them are processor dependent.

If a procedure is defined by means of a companion processor that is not the Fortran processor itself, this document refers to the C function that defines the procedure, although the procedure need not be defined by means of the
C programming language.

NOTE 5.14

A companion processor might or might not be a mechanism that conforms to the requirements of ISO/IEC 9899:2011. If it does, 5.3.7 states that a program unit that is defined by means other than Fortran and that initiates normal termination is required to include the effect of executing the C `exit()` function.

For example, a processor might allow a procedure defined by some language other than Fortran or C to be invoked if it can be described by a C prototype as defined in 6.7.6.3 of ISO/IEC 9899:2011.
6 Lexical tokens and source form

6.1 Processor character set

6.1.1 Characters

The processor character set is processor dependent. Each character in a processor character set is either a control character or a graphic character. The set of graphic characters is further divided into letters (6.1.2), digits (6.1.3), underscore (6.1.4), special characters (6.1.5), and other characters (6.1.6).

The letters, digits, underscore, and special characters make up the Fortran character set. Together, the set of letters, digits, and underscore define the syntax class alphanumeric-character.

R601 alphanumeric-character is letter or digit or underscore

Except for the currency symbol, the graphics used for the characters shall be as given in 6.1.2, 6.1.3, 6.1.4, and 6.1.5. However, the style of any graphic is not specified.

6.1.2 Letters

The twenty-six letters are:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

The set of letters defines the syntactic class letter. The processor character set shall include lower-case and upper-case letters. A lower-case letter is equivalent to the corresponding upper-case letter in program units except in a character context (3.21).

NOTE 6.1

The following statements are equivalent:

CALL BIG_COMPLEX_OPERATION (NDATE)
call big_complex_operation (ndate)
Call Big_Complex_Operation (NDate)

6.1.3 Digits

The ten digits are:

0 1 2 3 4 5 6 7 8 9

The ten digits define the syntactic class digit.

6.1.4 Underscore

R602 underscore is _

6.1.5 Special characters

The special characters are shown in Table 6.1.
6.1.6 Other characters

Additional characters may be representable in the processor, but may appear only in comments (6.3.2.3, 6.3.3.2), character constants (7.4.4), input/output records (12.2.2), and character string edit descriptors (13.3.2).

6.2 Low-level syntax

6.2.1 Tokens

The low-level syntax describes the fundamental lexical tokens of a program unit. A lexical token is a keyword, name, literal constant other than a complex literal constant, operator, statement label, delimiter, comma, =, =>, :, ::, ;, or %.

6.2.2 Names

Names are used for various entities such as variables, program units, dummy arguments, named constants, and derived types.

R603 name is letter [ alphanumerical-character ] ...

C601 (R603) The maximum length of a name is 63 characters.

NOTE 6.2

Examples of names:

A1 NAME_LENGTH (single underscore)
S_P_R_E_A_D__O_U_T (two consecutive underscores)
TRAILER_ (trailing underscore)
NOTE 6.3
The word “name” always denotes this particular syntactic form. The word “identifier” is used where entities
can be identified by other syntactic forms or by values; its particular meaning depends on the context in
which it is used.

6.2.3 Constants

R604 constant is literal-constant
or named-constant

R605 literal-constant is int-literal-constant
or real-literal-constant
or complex-literal-constant
or logical-literal-constant
or char-literal-constant
or boz-literal-constant

R606 named-constant is name

R607 int-constant is constant

C602 (R607) int-constant shall be of type integer.

6.2.4 Operators

R608 intrinsic-operator is power-op
or mult-op
or add-op
or concat-op
or rel-op
or not-op
or and-op
or or-op
or equiv-op

R1007 power-op is **

R1008 mult-op is *
or /

R1009 add-op is +
or –

R1011 concat-op is //

R1013 rel-op is .EQ.
or .NE.
or .LT.
or .LE.
or .GT.
or .GE.
or ==
or /=
or <
or <=
or >
or >=
6.2.5 Statement labels

A statement label provides a means of referring to an individual statement.

At least one digit in a label shall be nonzero.

If a statement is labeled, the statement shall contain a nonblank character. The same statement label shall not be given to more than one statement in its scope. Leading zeros are not significant in distinguishing between statement labels. There are 99999 possible unique statement labels and a processor shall accept any of them as a statement label. However, a processor may have a limit on the total number of unique statement labels in one program unit.

NOTE 6.4
For example:

99999
10
010

are all statement labels. The last two are equivalent.

Any statement that is not part of another statement, and that is not preceded by a semicolon in fixed form, may begin with a statement label, but the labels are used only in the following ways.

- The label on a branch target statement (11.2) is used to identify that statement as the possible destination of a branch.
- The label on a FORMAT statement (13.2.1) is used to identify that statement as the format specification for a data transfer statement (12.6).
- In some forms of the DO construct (11.1.7), the terminal statement of the construct is identified by a label.

6.2.6 Delimiters

A lexical token that is a delimiter is a (, ), /, [ ], (/ or /).
6.3 Source form

6.3.1 Program units, statements, and lines

1 A Fortran program unit is a sequence of one or more lines, organized as Fortran statements, comments, and INCLUDE lines. A line is a sequence of zero or more characters. Lines following a program unit END statement are not part of that program unit. A Fortran statement is a sequence of one or more complete or partial lines.

2 A comment may contain any character that may occur in any character context.

3 There are two source forms. Subclause 6.3.2 applies only to free form source. Subclause 6.3.3 applies only to fixed source form. Free form and fixed form shall not be mixed in the same program unit. The means for specifying the source form of a program unit are processor dependent.

6.3.2 Free source form

6.3.2.1 Free form line length

1 In free source form there are no restrictions on where a statement (or portion of a statement) may appear within a line. A line may contain zero characters. If a line consists entirely of characters of default kind (7.4.4), it may contain at most 132 characters. If a line contains any character that is not of default kind, the maximum number of characters allowed on the line is processor dependent.

6.3.2.2 Blank characters in free form

1 In free source form blank characters shall not appear within lexical tokens other than in a character context or in a format specification. Blanks may be inserted freely between tokens to improve readability; for example, blanks may occur between the tokens that form a complex literal constant. A sequence of blank characters outside of a character context is equivalent to a single blank character.

2 A blank shall be used to separate names, constants, or labels from adjacent keywords, names, constants, or labels.

NOTE 6.5 For example, the blanks after REAL, READ, 30, and DO are required in the following:

```plaintext
REAL X
READ 10
30 DO K=1,3
```

3 One or more blanks shall be used to separate adjacent keywords except in the following cases, where blanks are optional:

Table 6.2: Adjacent keywords where separating blanks are optional

<table>
<thead>
<tr>
<th>BLOCK DATA</th>
<th>END ENUM</th>
<th>END SELECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOUBLE PRECISION</td>
<td>END FILE</td>
<td>END SUBMODULE</td>
</tr>
<tr>
<td>ELSE IF</td>
<td>END FORALL</td>
<td>END SUBROUTINE</td>
</tr>
<tr>
<td>ELSE WHERE</td>
<td>END FUNCTION</td>
<td>END TYPE</td>
</tr>
<tr>
<td>END ASSOCIATE</td>
<td>END IF</td>
<td>END WHERE</td>
</tr>
<tr>
<td>END BLOCK</td>
<td>END INTERFACE</td>
<td>GO TO</td>
</tr>
<tr>
<td>END BLOCK DATA</td>
<td>END MODULE</td>
<td>IN OUT</td>
</tr>
<tr>
<td>END CRITICAL</td>
<td>END PROCEDURE</td>
<td>SELECT CASE</td>
</tr>
<tr>
<td>END DO</td>
<td>END PROGRAM</td>
<td>SELECT TYPE</td>
</tr>
</tbody>
</table>
6.3.2.3 Free form commentary

1. The character “!” initiates a comment except where it appears within a character context. The comment extends to the end of the line. If the first nonblank character on a line is an “!”, the line is a comment line. Lines containing only blanks or containing no characters are also comment lines. Comments may appear anywhere in a program unit and may precede the first statement of a program unit or follow the last statement of a program unit. Comments have no effect on the interpretation of the program unit.

NOTE 6.6

This document does not restrict the number of consecutive comment lines.

6.3.2.4 Free form statement continuation

1. The character “&” is used to indicate that the statement is continued on the next line that is not a comment line. Comment lines cannot be continued; an “&” in a comment has no effect. Comments may occur within a continued statement. When used for continuation, the “&” is not part of the statement. No line shall contain a single “&” as the only nonblank character or as the only nonblank character before an “!” that initiates a comment.

2. If a noncharacter context is to be continued, an “&” shall be the last nonblank character on the line, or the last nonblank character before an “!”. There shall be a later line that is not a comment; the statement is continued on the next such line. If the first nonblank character on that line is an “&”, the statement continues at the next character position following that “&”; otherwise, it continues with the first character position of that line.

3. If a lexical token is split across the end of a line, the first nonblank character on the first following noncomment line shall be an “&” immediately followed by the successive characters of the split token.

4. If a character context is to be continued, an “&” shall be the last nonblank character on the line. There shall be a later line that is not a comment; an “&” shall be the first nonblank character on the next such line and the statement continues with the next character following that “&”.

6.3.2.5 Free form statement termination

1. If a statement is not continued, a comment or the end of the line terminates the statement.

2. A statement may alternatively be terminated by a “;” character that appears other than in a character context or in a comment. The “;” is not part of the statement. After a “;” terminator, another statement may appear on the same line, or begin on that line and be continued. A sequence consisting only of zero or more blanks and one or more “;” terminators, in any order, is equivalent to a single “;” terminator.

6.3.2.6 Free form statements

1. A label may precede any statement not forming part of another statement.

NOTE 6.7

No Fortran statement begins with a digit.

2. A statement shall not have more than 255 continuation lines.

6.3.3 Fixed source form

6.3.3.1 General

1. In fixed source form, there are restrictions on where a statement may appear within a line. If a source line contains only characters of default kind, it shall contain exactly 72 characters; otherwise, its maximum number of characters is processor dependent.

2. Except in a character context, blanks are insignificant and may be used freely throughout the program.
6.3.3.2 Fixed form commentary

The character "!" initiates a comment except where it appears within a character context or in character position 6. The comment extends to the end of the line. If the first nonblank character on a line is an "!" in any character position other than character position 6, the line is a comment line. Lines beginning with a "C" or "*=" in character position 1 and lines containing only blanks are also comment lines. Comments may appear anywhere in a program unit and may precede the first statement of the program unit or follow the last statement of a program unit. Comments have no effect on the interpretation of the program unit.

NOTE 6.8

This document does not restrict the number of consecutive comment lines.

6.3.3.3 Fixed form statement continuation

Except within commentary, character position 6 is used to indicate continuation. If character position 6 contains a blank or zero, the line is the initial line of a new statement, which begins in character position 7. If character position 6 contains any character other than blank or zero, character positions 7–72 of the line constitute a continuation of the preceding noncomment line.

NOTE 6.9

An "!" or ";" in character position 6 is interpreted as a continuation indicator unless it appears within commentary indicated by a "C" or "*=" in character position 1 or by an "!" in character positions 1–5.

Comment lines cannot be continued. Comment lines may occur within a continued statement.

6.3.3.4 Fixed form statement termination

If a statement is not continued, a comment or the end of the line terminates the statement.

A statement may alternatively be terminated by a ";" character that appears other than in a character context, in a comment, or in character position 6. The ";" is not part of the statement. After a ";" terminator, another statement may begin on the same line, or begin on that line and be continued. A ";" shall not appear as the first nonblank character on an initial line. A sequence consisting only of zero or more blanks and one or more ";" terminators, in any order, is equivalent to a single ";" terminator.

6.3.3.5 Fixed form statements

A label, if it appears, shall occur in character positions 1 through 5 of the first line of a statement; otherwise, positions 1 through 5 shall be blank. Blanks may appear anywhere within a label. A statement following a ";" on the same line shall not be labeled. Character positions 1 through 5 of any continuation lines shall be blank. A statement shall not have more than 255 continuation lines. The program unit END statement shall not be continued. A statement whose initial line appears to be a program unit END statement shall not be continued.

6.4 Including source text

Additional text may be incorporated into the source text of a program unit during processing. This is accomplished with the INCLUDE line, which has the form

INCLUDE char-literal-constant

The char-literal-constant shall not have a kind type parameter value that is a named-constant.

An INCLUDE line is not a Fortran statement.

An INCLUDE line shall appear on a single source line where a statement may appear; it shall be the only nonblank text on this line other than an optional trailing comment. Thus, a statement label is not allowed.

The effect of the INCLUDE line is as if the referenced source text physically replaced the INCLUDE line prior to program processing. Included text may contain any source text, including additional INCLUDE lines; such nested INCLUDE lines are similarly replaced with the specified source text. The maximum depth of nesting of any nested INCLUDE lines is processor dependent. Inclusion of the source text referenced by an INCLUDE line shall not, at any level of nesting, result in inclusion of the same source text.
When an INCLUDE line is resolved, the first included statement line shall not be a continuation line and the last included statement line shall not be continued.

The interpretation of \textit{char-literal-constant} is processor dependent. An example of a possible valid interpretation is that \textit{char-literal-constant} is the name of a file that contains the source text to be included.

\begin{note}

In some circumstances, for example where source code is maintained in an INCLUDE file for use in programs whose source form might be either fixed or free, observing the following rules allows the code to be used with either source form.

- Confine statement labels to character positions 1 to 5 and statements to character positions 7 to 72.
- Treat blanks as being significant.
- Use only the exclamation mark (!) to indicate a comment, but do not start the comment in character position 6.
- For continued statements, place an ampersand (&) in both character position 73 of a continued line and character position 6 of a continuation line.

\end{note}
7 Types

7.1 Characteristics of types

7.1.1 The concept of type
1 Fortran provides an abstract means whereby data can be categorized without relying on a particular physical representation. This abstract means is the concept of type.
2 A 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.

7.1.2 Type classification
1 A type is either an intrinsic type or a derived type.
2 This document defines five intrinsic types: integer, real, complex, character, and logical.
3 A derived type is one that is defined by a derived-type definition (7.5.2) or by an intrinsic module. It shall be used only where it is accessible (7.5.2.2). An intrinsic type is always accessible.

7.1.3 Set of values
1 For each type, there is a set of valid values. The set of valid values for logical is completely determined by this document. The sets of valid values for integer, character, and real are processor dependent. The set of valid values for complex consists of the set of all the combinations of the values of the real and imaginary parts. The set of valid values for a derived type is as defined in 7.5.8.

7.1.4 Constants
1 The syntax for denoting a value indicates the type, type parameters, and the particular value.
2 The syntax for literal constants of each intrinsic type is specified in 7.4.
3 A structure constructor (7.5.10) that is a constant expression (10.1.12) denotes a scalar constant value of derived type. An array constructor (7.8) that is a constant expression denotes a constant array value of intrinsic or derived type.
4 A constant value can be named (8.5.13, 8.6.11).

7.1.5 Operations
1 For each of the intrinsic types, a set of operations and corresponding operators is defined intrinsically. These are described in Clause 10. The intrinsic set can be augmented with operations and operators defined by functions with the OPERATOR interface (15.4.3.2). Operator definitions are described in Clauses 10 and 15.
2 For derived types, there are no intrinsic operations. Operations on derived types can be defined by the program (7.5.11).
1 7.2 Type parameters
2
3 A type might be parameterized. In this case, the set of values, the syntax for denoting the values, and the set of
4 operations on the values of the type depend on the values of the parameters.
5
6 A type parameter is either a kind type parameter or a length type parameter. All type parameters are of type
7 integer. A kind type parameter participates in generic resolution (15.5.5.2), but a length type parameter does
8 not.
9
10 Each intrinsic type has a kind type parameter named KIND. The intrinsic character type has a length type
11 parameter named LEN. A derived type can have type parameters.
12
13 A type parameter value may be specified by a type specification (7.4, 7.5.9).
14
15 R701 type-param-value is scalar-int-expr
16 or *
17 or :
18
19 C701 (R701) The type-param-value for a kind type parameter shall be a constant expression.
20
21 C702 (R701) A colon shall not be used as a type-param-value except in the declaration of an entity that has
22 the POINTER or ALLOCATABLE attribute.
23
24 A colon as a type-param-value specifies a deferred type parameter.
25
26 The values of the deferred type parameters of an object are determined by successful execution of an ALLOCATE
27 statement (9.7.1), execution of an intrinsic assignment statement (10.2.1.3), execution of a pointer assignment
28 statement (10.2.2), or by argument association (15.5.2).
29
NOTE 7.1
Deferred type parameters of functions, including function procedure pointers, have no values. Instead, they
indicate that those type parameters of the function result will be determined by execution of the function,
if it returns an allocated allocatable result or an associated pointer result.
30
31 An asterisk as a type-param-value specifies that a length type parameter is an assumed type parameter. It is used
32 for a dummy argument to assume the type parameter value from the effective argument, for an associate name
33 in a SELECT TYPE construct to assume the type parameter value from the corresponding selector, and for a
34 named constant of type character to assume the character length from the constant-expr.
35
NOTE 7.2
The value of a kind type parameter is always known at compile time. Some parameterizations that involve
multiple representation forms need to be distinguished at compile time for practical implementation and
performance. Examples include the multiple precisions of the intrinsic real type and the possible multiple
character sets of the intrinsic character type.

The adjective “length” is used for type parameters other than kind type parameters because they often
specify a length, as for intrinsic character type. However, they can be used for other purposes. The
important difference from kind type parameters is that their values need not be known at compile time and
might change during execution.

36 7.3 Types, type specifiers, and values
37
38 7.3.1 Relationship of types and values to objects
39
40 The name of a type serves as a type specifier and may be used to declare objects of that type. A declaration
41 specifies the type of a named object. A data object may be declared explicitly or implicitly. A data object has
attributes in addition to its type. Clause 8 describes the way in which a data object is declared and how its type and other attributes are specified.

Scalar data of any intrinsic or derived type may be shaped in a rectangular pattern to compose an array of the same type and type parameters. An array object has a type and type parameters just as a scalar object does.

A variable is a data object. The type and type parameters of a variable determine which values that variable may take. Assignment (10.2) provides one means of changing the value of a variable of any type.

7.3.2 Type specifiers and type compatibility

7.3.2.1 Type specifier syntax

1 A type specifier specifies a type and type parameter values. It is either a type-spec or a declaration-type-spec.

R702 type-spec is intrinsic-type-spec
or derived-type-spec

C703 (R702) The derived-type-spec shall not specify an abstract type (7.5.7).

R703 declaration-type-spec is intrinsic-type-spec
or TYPE ( intrinsic-type-spec )
or TYPE ( derived-type-spec )
or CLASS ( derived-type-spec )
or CLASS ( * )
or TYPE ( * )

C704 (R703) In a declaration-type-spec, every type-param-value that is not a colon or an asterisk shall be a specification-expr.

C705 (R703) In a declaration-type-spec that uses the CLASS keyword, derived-type-spec shall specify an extensible type (7.5.7).

C706 (R703) TYPE(derived-type-spec) shall not specify an abstract type (7.5.7).

C707 (R702) In TYPE(intrinsic-type-spec) the intrinsic-type-spec shall not end with a comma.

C708 An entity declared with the CLASS keyword shall be a dummy argument or have the ALLOCATABLE or POINTER attribute.

2 An intrinsic-type-spec specifies the named intrinsic type and its type parameter values. A derived-type-spec specifies the named derived type and its type parameter values.

NOTE 7.3
A type-spec is used in an array constructor, a SELECT TYPE construct, or an ALLOCATE statement. An integer-type-spec is used in a DO CONCURRENT or FORALL statement. Elsewhere, a declaration-type-spec is used.

7.3.2.2 TYPE type specifier

1 A TYPE type specifier is used to declare entities that are assumed-type, or of an intrinsic or derived type.

2 A derived-type-spec in a TYPE type specifier in a type declaration statement shall specify a previously defined derived type. If the data entity is a function result, the derived type may be specified in the FUNCTION statement provided the derived type is defined within the body of the function or is accessible there by use or host association. If the derived type is specified in the FUNCTION statement and is defined within the body of the function, it is as if the function result were declared with that derived type immediately following the
`derived-type-def` of the specified derived type.

An entity that is declared using the `TYPE(*)` type specifier is assumed-type and is an unlimited polymorphic entity. It is not declared to have a type, and is not considered to have the same declared type as any other entity, including another unlimited polymorphic entity. Its dynamic type and type parameters are assumed from its effective argument.

C709 An assumed-type entity shall be a dummy data object that does not have the ALLOCATABLE, CODIMENSION, INTENT (OUT), POINTER, or VALUE attribute and is not an explicit-shape array.

C710 An assumed-type variable name shall not appear in a designator or expression except as an actual argument corresponding to a dummy argument that is assumed-type, or as the first argument to the intrinsic function IS_CONTIGUOUS, LBOUND, PRESENT, RANK, SHAPE, SIZE, or UBOUND, or the function C_LOC from the intrinsic module ISO_C_BINDING.

C711 An assumed-type actual argument that corresponds to an assumed-rank dummy argument shall be assumed-shape or assumed-rank.

7.3.2.3 CLASS type specifier

The CLASS type specifier is used to declare polymorphic entities. A polymorphic entity is a data entity that is able to be of differing dynamic types during program execution.

A `derived-type-spec` in a CLASS type specifier in a type declaration statement shall specify a previously defined derived type. If the data entity is a function result, the derived type may be specified in the FUNCTION statement provided the derived type is defined within the body of the function or is accessible there by use or host association. If the derived type is specified in the FUNCTION statement and is defined within the body of the function, it is as if the function result were declared with that derived type immediately following its `derived-type-def`.

The declared type of a polymorphic entity is the specified type if the CLASS type specifier contains a type name.

An entity declared with the CLASS(*) specifier is an unlimited polymorphic entity. It is not declared to have a type, and is not considered to have the same declared type as any other entity, including another unlimited polymorphic entity.

A nonpolymorphic entity is type compatible only with entities of the same declared type. A polymorphic entity that is not an unlimited polymorphic entity is type compatible with entities of the same declared type or any of its extensions. Even though an unlimited polymorphic entity is not considered to have a declared type, it is type compatible with all entities. An entity is type compatible with a type if it is type compatible with entities of that type.

**NOTE 7.4**

Given

```fortran
TYPE TROOT
  ...
  TYPE, EXTENDS(TROOT) :: TEXTENDED
  ...
CLASS(TROOT) A
CLASS(TEXTENDED) B
  ...
```

A is type compatible with B but B is not type compatible with A.

A polymorphic allocatable object may be allocated to be of any type with which it is type compatible. A polymorphic pointer or dummy argument may, during program execution, be associated with objects with which it is type compatible.
The dynamic type of an allocated allocatable polymorphic object is the type with which it was allocated. The
dynamic type of an associated polymorphic pointer is the dynamic type of its target. The dynamic type of a
nonallocatable nonpointer polymorphic dummy argument is the dynamic type of its effective argument. The
dynamic type of an unallocated allocatable object or a disassociated pointer is the same as its declared type. The
dynamic type of an entity identified by an associate name (11.1.3) is the dynamic type of the selector with which
it is associated. The dynamic type of an object that is not polymorphic is its declared type.

7.4 Intrinsic types

7.4.1 Classification and specification

1 Each intrinsic type is classified as a numeric type or a nonnumeric type. The numeric types are integer, real, and
complex. The nonnumeric intrinsic types are character and logical.

2 Each intrinsic type has a kind type parameter named KIND; this type parameter is of type integer with default
kind.

R704 intrinsic-type-spec is integer-type-spec
or REAL [ kind-selector ]
or DOUBLE PRECISION
or COMPLEX [ kind-selector ]
or CHARACTER [ char-selector ]
or LOGICAL [ kind-selector ]

R705 integer-type-spec is INTEGER [ kind-selector ]
R706 kind-selector is ( [ KIND = ] scalar-int-constant-expr )

C712 (R706) The value of scalar-int-constant-expr shall be nonnegative and shall specify a representation
method that exists on the processor.

7.4.2 Intrinsic operations on intrinsic types

1 Intrinsic numeric operations are defined as specified in 10.1.5.2.1 for the numeric intrinsic types. Relational
intrinsic operations are defined as specified in 10.1.5.5 for numeric and character intrinsic types. The intrinsic
concatenation operation is defined as specified in 10.1.5.3 for the character type. Logical intrinsic operations are
defined as specified in 10.1.5.4 for the logical type.

7.4.3 Numeric intrinsic types

7.4.3.1 Integer type

1 The set of values for the integer type is a subset of the mathematical integers. The processor shall provide one or
more representation methods that define sets of values for data of type integer. Each such method is characterized
by a value for the kind type parameter KIND. The kind type parameter of a representation method is returned
by the intrinsic function KIND (16.9.108). The decimal exponent range of a representation method is returned
by the intrinsic function RANGE (16.9.158). The intrinsic function SELECTED_INT_KIND (16.9.169) returns
a kind value based on a specified decimal exponent range requirement. The integer type includes a zero value,
which is considered to be neither negative nor positive. The value of a signed integer zero is the same as the
value of an unsigned integer zero.

2 The processor shall provide at least one representation method with a decimal exponent range greater than or
equal to 18.

3 The type specifier for the integer type uses the keyword INTEGER.
The keyword INTEGER with no kind-selector specifies type integer with default kind; the kind type parameter value is equal to KIND (0). The decimal exponent range of default integer shall be at least 5.

Any integer value may be represented as a signed-int-literal-constant.

R707 signed-int-literal-constant is [ sign ] int-literal-constant
R708 int-literal-constant is digit-string [ _ kind-param ]
R709 kind-param is digit-string or scalar-int-constant-name
R710 signed-digit-string is [ sign ] digit-string
R711 digit-string is digit [ digit ] ...
R712 sign is + or –

C713 (R709) A scalar-int-constant-name shall be a named constant of type integer.
C714 (R709) The value of kind-param shall be nonnegative.
C715 (R708) The value of kind-param shall specify a representation method that exists on the processor.

The optional kind type parameter following digit-string specifies the kind type parameter of the integer constant; if it is does not appear, the constant is default integer.

An integer constant is interpreted as a decimal value.

**NOTE 7.5**

Examples of signed integer literal constants are:

473
+56
-101
21_2
21_SHORT
1976354279568241_8

where SHORT is a scalar integer named constant. A program that uses a digit-string as a kind-param is unlikely to be portable.

**7.4.3.2 Real type**

1 The real type has values that approximate the mathematical real numbers. The processor shall provide two or more approximation methods that define sets of values for data of type real. Each such method has a representation method and is characterized by a value for the kind type parameter KIND. The kind type parameter of an approximation method is returned by the intrinsic function KIND (16.9.108).

2 The decimal precision, decimal exponent range, and radix of an approximation method are returned by the intrinsic functions PRECISION (16.9.151), RADIX (16.9.154) and RANGE (16.9.158). The intrinsic function SELECTED_REAL_KIND (16.9.170) returns a kind value based on specified precision, range, and radix requirements.

**NOTE 7.6**

See C.2.1 for remarks concerning selection of approximation methods.
The real type includes a zero value. Processors that distinguish between positive and negative zeros shall treat them as mathematically equivalent:

- in all intrinsic relational operations, and
- as actual arguments to intrinsic procedures other than those for which it is explicitly specified that negative zero is distinguished.

NOTE 7.7

On a processor that distinguishes between 0.0 and −0.0,

\[(X \geq 0.0)\]
evaluates to true if \(X = 0.0\) or if \(X = -0.0\),

\[(X < 0.0)\]
evaluates to false for \(X = -0.0\).

In order to distinguish between 0.0 and −0.0, a program can use the intrinsic function SIGN. SIGN (1.0, X) will return −1.0 if \(X < 0.0\) or if the processor distinguishes between 0.0 and −0.0 and X has the value −0.0.

The type specifier for the real type uses the keyword REAL. The keyword DOUBLE PRECISION is an alternative specifier for one kind of real type.

If the type keyword REAL is used without a kind type parameter, the real type with default real kind is specified and the kind value is KIND (0.0). The type specifier DOUBLE PRECISION specifies type real with double precision kind; the kind value is KIND (0.0D0). The decimal precision of the double precision real approximation method shall be greater than that of the default real method.

The decimal precision of double precision real shall be at least 10, and its decimal exponent range shall be at least 37. It is recommended that the decimal precision of default real be at least 6, and that its decimal exponent range be at least 37.

If both kind-param and exponent-letter appear, exponent-letter shall be E.

The value of kind-param shall specify an approximation method that exists on the processor.

A real literal constant without a kind type parameter is a default real constant if it is without an exponent part or has exponent letter E, and is a double precision real constant if it has exponent letter D. A real literal constant written with a kind type parameter is a real constant with the specified kind type parameter.

The exponent represents the power of ten scaling to be applied to the significand or digit string. The meaning of these constants is as in decimal scientific notation.

The significand may be written with more digits than a processor will use to approximate the value of the constant.
NOTE 7.8

Examples of signed real literal constants are:

-12.78
+1.6E3
2.1
-16.E4_8
0.45D-4
10.93E7_QUAD
.123
3E4

where QUAD is a scalar integer named constant.

7.4.3.3 Complex type

1 The complex type has values that approximate the mathematical complex numbers. The values of a complex
type are ordered pairs of real values. The first real value is called the real part, and the second real value is called
the imaginary part.

2 Each approximation method used to represent data entities of type real shall be available for both the real and
imaginary parts of a data entity of type complex. The (default integer) kind type parameter KIND for a complex
entity specifies for both parts the real approximation method characterized by this kind type parameter value.
The kind type parameter of an approximation method is returned by the intrinsic function KIND (16.9.108).

3 The type specifier for the complex type uses the keyword COMPLEX. There is no keyword for double precision
complex. If the type keyword COMPLEX is used without a kind type parameter, the complex type with default
complex kind is specified, the kind value is KIND (0.0), and both parts are default real.

R718 complex-literal-constant is ( real-part , imag-part )
R719 real-part is signed-int-literal-constant
or signed-real-literal-constant
or named-constant
R720 imag-part is signed-int-literal-constant
or signed-real-literal-constant
or named-constant

C718 (R718) Each named constant in a complex literal constant shall be of type integer or real.

4 If the real part and the imaginary part of a complex literal constant are both real, the kind type parameter value
of the complex literal constant is the kind type parameter value of the part with the greater decimal precision; if
the precisions are the same, it is the kind type parameter value of one of the parts as determined by the processor.
If a part has a kind type parameter value different from that of the complex literal constant, the part is converted
to the approximation method of the complex literal constant.

5 If both the real and imaginary parts are integer, they are converted to the default real approximation method
and the constant is default complex. If only one of the parts is an integer, it is converted to the approximation
method selected for the part that is real and the kind type parameter value of the complex literal constant is
that of the part that is real.

NOTE 7.9

Examples of complex literal constants are:
### 7.4.4 Character type

#### 7.4.4.1 Character sets

1. The character type has a set of values composed of character strings. A character string is a sequence of characters, numbered from left to right 1, 2, 3, ... up to the number of characters in the string. The number of characters in the string is called the length of the string. The length is a type parameter; its kind is processor dependent and its value is greater than or equal to zero.

2. The processor shall provide one or more representation methods that define sets of values for data of type character. Each such method is characterized by a value for the (default integer) kind type parameter KIND. The kind type parameter of a representation method is returned by the intrinsic function KIND (16.9.108). The intrinsic function SELECTED_CHAR_KIND (16.9.168) returns a kind value based on the name of a character type. Any character of a particular representation method representable in the processor may occur in a character string of that representation method.

3. The character set specified in ISO/IEC 646:1991 (International Reference Version) is referred to as the ASCII character set and its corresponding representation method is ASCII character kind. The character set UCS-4 as specified in ISO/IEC 10646 is referred to as the ISO 10646 character set and its corresponding representation method is the ISO 10646 character kind.

#### 7.4.4.2 Character type specifier

1. The type specifier for the character type uses the keyword CHARACTER.

2. If the type keyword CHARACTER is used without a kind type parameter, the character type with default character kind is specified and the kind value is KIND (‘A’).

3. The default character kind shall support a character set that includes the characters in the Fortran character set (6.1). By supplying nondefault character kinds, the processor may support additional character sets. The characters available in nondefault character kinds are not specified by this document, except that one character in each nondefault character set shall be designated as a blank character to be used as a padding character.

<table>
<thead>
<tr>
<th>Note</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R721</td>
<td>char-selector is length-selector or ( LEN = type-param-value , KIND = scalar-int-constant-expr ) or ( type-param-value , [ KIND = scalar-int-constant-expr ] )</td>
</tr>
<tr>
<td>R722</td>
<td>length-selector is ( [ LEN = type-param-value ] ) or * char-length [ , ]</td>
</tr>
<tr>
<td>R723</td>
<td>char-length is ( type-param-value ) or int-literal-constant</td>
</tr>
<tr>
<td>C719</td>
<td>(R721) The value of scalar-int-constant-expr shall be nonnegative and shall specify a representation method that exists on the processor.</td>
</tr>
<tr>
<td>C720</td>
<td>(R723) The int-literal-constant shall not include a kind-param.</td>
</tr>
</tbody>
</table>
C721 (R723) A type-param-value in a char-length shall be a colon, asterisk, or specification-expr.

C722 (R721 R722 R723) A type-param-value of * shall be used only

• to declare a dummy argument,
• to declare a named constant,
• in the type-spec of an ALLOCATE statement wherein each allocate-object is a dummy argument of type CHARACTER with an assumed character length,
• in the type-spec or derived-type-spec of a type guard statement (11.1.11), or
• in an external function, to declare the character length parameter of the function result.

C723 A function name shall not be declared with an asterisk type-param-value unless it is of type CHARACTER and is the name of a dummy function or the name of the result of an external function.

C724 A function name declared with an asterisk type-param-value shall not be an array, a pointer, elemental, recursive, or pure.

C725 (R722) The optional comma in a length-selector is permitted only in a declaration-type-spec in a type-declaration-stmt.

C726 (R722) The optional comma in a length-selector is permitted only if no double-colon separator appears in the type-declaration-stmt.

C727 (R721) The length specified for a character statement function or for a statement function dummy argument of type character shall be a constant expression.

4 The char-selector in a CHARACTER intrinsic-type-spec and the * char-length in an entity-decl or in a component-decl of a type definition specify character length. The * char-length in an entity-decl or a component-decl specifies an individual length and overrides the length specified in the char-selector, if any. If a * char-length is not specified in an entity-decl or a component-decl, the length-selector or type-param-value specified in the char-selector is the character length. If the length is not specified in a char-selector or a * char-length, the length is 1.

5 If the character length parameter value evaluates to a negative value, the length of character entities declared is zero. A character length parameter value of : indicates a deferred type parameter (7.2). A char-length type parameter value of * has the following meanings.

• If used to declare a dummy argument of a procedure, the dummy argument assumes its length from its effective argument.
• If used to declare a named constant, the length is that of the constant value.
• If used in the type-spec of an ALLOCATE statement, each allocate-object assumes its length from its effective argument.
• If used in the type-spec of a type guard statement, the associating entity assumes its length from the selector.
• If used to specify the character length parameter of a function result, any scoping unit invoking the function or passing it as an actual argument shall declare the function name with a character length parameter value other than * or access such a definition by argument, host, or use association. When the function is invoked, the length of the function result is assumed from the value of this type parameter.

7.4.4.3 Character literal constant
1 The syntax of a character literal constant is given by R724.

R724 char-literal-constant is [ kind-param _ ] ' [ rep-char ] ... ' or [ kind-param _ ] " [ rep-char ] ... "

C728 (R724) The value of kind-param shall specify a representation method that exists on the processor.

2 The optional kind type parameter preceding the leading delimiter specifies the kind type parameter of the character constant; if it does not appear, the constant is default character.
For the type character with kind \textit{kind-param}, if it appears, and for default character otherwise, a representable character, \textit{rep-char}, is defined as follows.

- In free source form, it is any graphic character in the processor-dependent character set.
- In fixed source form, it is any character in the processor-dependent character set. A processor may restrict the occurrence of some or all of the control characters.

The delimiting apostrophes or quotation marks are not part of the value of the character literal constant.

An apostrophe character within a character constant delimited by apostrophes is represented by two consecutive apostrophes (without intervening blanks); in this case, the two apostrophes are counted as one character. Similarly, a quotation mark character within a character constant delimited by quotation marks is represented by two consecutive quotation marks (without intervening blanks) and the two quotation marks are counted as one character.

A zero-length character literal constant is represented by two consecutive apostrophes (without intervening blanks) or two consecutive quotation marks (without intervening blanks) outside of a character context.

\textbf{NOTE 7.10}

Examples of character literal constants are:

"DON'T"

'DON'T'

both of which have the value DON'T and

'',

which has the zero-length character string as its value.

\textbf{NOTE 7.11}

An example of a nondefault character literal constant, where the processor supports the corresponding character set, is:

NIHONGO_'彼女なしでは何もできない' 

where NIHONGO is a named constant whose value is the kind type parameter for Nihongo (Japanese) characters. This means “Without her, nothing is possible”.

7.4.4.4 Collating sequence

The processor defines a collating sequence for the character set of each kind of character. The collating sequence is an isomorphism between the character set and the set of integers \( \{ I : 0 \leq I < N \} \), where \( N \) is the number of characters in the set. The intrinsic functions \texttt{CHAR} (16.9.44) and \texttt{ICHAR} (16.9.95) provide conversions between the characters and the integers according to this mapping.

\textbf{NOTE 7.12}

For example:

\texttt{ICHAR ('X')}

returns the integer value of the character 'X' according to the collating sequence of the processor.

The collating sequence of the default character kind shall satisfy the following constraints.

- \texttt{ICHAR ('A')} < \texttt{ICHAR ('B')} < ... < \texttt{ICHAR ('Z')} for the twenty-six upper-case letters.
• ICHAR (‘0’) < ICHAR (‘1’) < ... < ICHAR (‘9’) for the ten digits.
• ICHAR (‘ ‘) < ICHAR (‘0’) < ICHAR (‘9’) < ICHAR (‘A’) or
  ICHAR (‘ ‘) < ICHAR (‘A’) < ICHAR (‘Z’) < ICHAR (‘0’).
• ICHAR (‘a’) < ICHAR (‘b’) < ... < ICHAR (‘z’) for the twenty-six lower-case letters.
• ICHAR (‘ ‘) < ICHAR (‘0’) < ICHAR (‘9’) < ICHAR (‘a’) or
  ICHAR (‘ ‘) < ICHAR (‘a’) < ICHAR (‘z’) < ICHAR (‘0’).

3 There are no constraints on the location of any other character in the **collating sequence**, nor is there any specified **collating sequence** relationship between the upper-case and lower-case letters.

4 The **collating sequence** for the **ASCII character kind** is as specified in ISO/IEC 646:1991 (International Reference Version); this **collating sequence** is called the **ASCII collating sequence** in this document. The **collating sequence** for the **ISO 10646 character kind** is as specified in ISO/IEC 10646.

5 The intrinsic functions ACHAR (16.9.3) and IACHAR (16.9.88) provide conversions between characters and corresponding integer values according to the **ASCII collating sequence**.

6 The intrinsic functions LGT, LGE, LLE, and LLT (16.9.114-16.9.117) provide comparisons between strings based on the **ASCII collating sequence**. International portability is guaranteed if the set of characters used is limited to the Fortran character set (6.1).

### 7.4.5 Logical type

1 The logical type has two values, which represent true and false.

2 The processor shall provide one or more representation methods for data of type logical. Each such method is characterized by a value for the (default integer) **kind type parameter** KIND. The **kind type parameter** of a representation method is returned by the intrinsic function KIND (16.9.108).

3 The type specifier for the logical type uses the keyword LOGICAL.

4 The keyword LOGICAL with no **kind-selector** specifies type logical with default kind; the **kind type parameter** value is equal to KIND (.FALSE.).

5 The optional **kind type parameter** specifies the **kind type parameter** of the logical constant; if it does not appear, the constant has the default logical kind.

### 7.5 Derived types

#### 7.5.1 Derived type concepts

1 Additional types can be derived from the intrinsic types and other derived types. A type definition defines the name of the type and the names and attributes of its components and **type-bound procedures**.

2 A derived type can be parameterized by one or more type parameters, each of which is defined to be either a **kind** or length type parameter and can have a default value.

3 The ultimate components of a derived type are the components that are of intrinsic type or have the ALLOCATABLE or POINTER attribute, plus the ultimate components of the components that are of derived type and have neither the ALLOCATABLE nor POINTER attribute.
The direct components of a derived type are the components of that type, plus the direct components of the
components that are of derived type and have neither the ALLOCATABLE nor POINTER attribute.

The potential subobject components of a derived type are the nonpointer components of that type together with
the potential subobject components of the nonpointer components that are of derived type. This includes all the
components that could be a subobject of an object of the type (9.4.2).

The components, direct components, potential subobject components, and ultimate components of an object of
derived type are the components, direct components, potential subobject components, and ultimate components
of its type, respectively.

By default, no storage sequence is implied by the order of the component definitions. However, a storage sequence
is implied for a sequence type (7.5.2.3). If the derived type has the BIND attribute, the storage sequence is that
required by the companion processor (5.5.7, 18.3.4).

A scalar entity of derived type is a structure. If a derived type has the SEQUENCE attribute, a scalar entity of
the type is a sequence structure.

NOTE 7.14
The ultimate components of an object of the derived type kids defined below are oldest_child%name,
oldest_child%age, and other_kids. The direct components of such an object are oldest_child%name,
oldest_child%age, other_kids, and oldest_child.

```
type :: person
  character(len=20) :: name
  integer :: age
end type person

type :: kids
  type(person) :: oldest_child
  type(person), allocatable, dimension(:) :: other_kids
end type kids
```

7.5.2 Derived-type definition

7.5.2.1 Syntax

R726 derived-type-def is derived-type-stmt

[ type-param-def-stmt ] ...
[ private-or-sequence ] ...
[ component-part ]
[ type-bound-procedure-part ]
end-type-stmt

R727 derived-type-stmt is TYPE [ [ , type-attr-spec-list ] :: ] type-name ■

■ [ ( type-param-name-list ) ]

R728 type-attr-spec is ABSTRACT

or access-spec

or BIND (C)

or EXTENDS ( parent-type-name )

C730 (R727) A derived type type-name shall not be DOUBLEPRECISION or the same as the name of any
intrinsic type defined in this document.

C731 (R727) The same type-attr-spec shall not appear more than once in a given derived-type-stmt.

C732 (R727) The same type-param-name shall not appear more than once in a derived-type-stmt.
A parent-type-name shall be the name of a previously defined extensible type (7.5.7).

If the type definition contains or inherits (7.5.7.2) a deferred type-bound procedure (7.5.5), ABSTRACT shall appear.

If ABSTRACT appears, the type shall be extensible.

If EXTENDS appears, SEQUENCE shall not appear.

If EXTENDS appears and the type being defined has a coarray ultimate component, its parent type shall have a coarray ultimate component.

If EXTENDS appears and the type being defined has a potential subobject component of type EVENT_TYPE or LOCK_TYPE from the intrinsic module ISO_FORTRAN_ENV, its parent type shall be EVENT_TYPE or LOCK_TYPE or have a potential subobject component of type EVENT_TYPE or LOCK_TYPE.

privatesequence is private-components-stmt or sequence-stmt

The same privatesequence shall not appear more than once in a given derived-type-def.

end-type-stmt is END TYPE [ type-name ]

If END TYPE is followed by a type-name, the type-name shall be the same as that in the corresponding derived-type-stmt.

Derived types with the BIND attribute are subject to additional constraints as specified in 18.3.4.

NOTE 7.15

An example of a derived-type definition is:

```
TYPE PERSON
  INTEGER AGE
  CHARACTER (LEN = 50) NAME
END TYPE PERSON
```

An example of declaring a variable CHAIRMAN of type PERSON is:

```
TYPE (PERSON) :: CHAIRMAN
```

7.5.2.2 Accessibility

The accessibility of a type name is determined as specified in 8.5.2. The accessibility of a type name does not affect, and is not affected by, the accessibility of its components and type-bound procedures.

If a derived type is defined in the scoping unit of a module, and its name is private in that module, then the type name, and thus the structure constructor (7.5.10) for the type, are accessible only within that module and its descendants.

NOTE 7.16

An example of a type with a private name is:

```
TYPE, PRIVATE :: AUXILIARY
  LOGICAL :: DIAGNOSTIC
  CHARACTER (LEN = 20) :: MESSAGE
END TYPE AUXILIARY
```
NOTE 7.16 (cont.)
Such a type would be accessible only within the module in which it is defined, and within its descendants.

7.5.2.3 Sequence type

1 R731 sequence-stmt is SEQUENCE

C741 (R726) If SEQUENCE appears, the type shall have at least one component, each data component shall be declared to be of an intrinsic type or of a sequence type, the derived type shall not have any type parameter, and a type-bound-procedure-part shall not appear.

1 If the SEQUENCE statement appears, the type has the SEQUENCE attribute and is a sequence type. The order of the component definitions in a sequence type specifies a storage sequence for objects of that type. The type is a numeric sequence type if there are no pointer or allocatable components, and each component is default integer, default real, double precision real, default complex, default logical, or of numeric sequence type. The type is a character sequence type if there are no pointer or allocatable components, and each component is default character or of character sequence type.

NOTE 7.17
An example of a numeric sequence type is:

TYPE NUMERIC_SEQ
  SEQUENCE
    INTEGER :: INT_VAL
    REAL :: REAL_VAL
    LOGICAL :: LOG_VAL
END TYPE NUMERIC_SEQ

NOTE 7.18
A structure resolves into a sequence of components. Unless the structure includes a SEQUENCE statement, the use of this terminology in no way implies that these components are stored in this, or any other, order. Nor is there any requirement that contiguous storage be used. The sequence merely refers to the fact that in writing the definitions there will necessarily be an order in which the components appear, and this will define a sequence of components. This order is of limited significance because a component of an object of derived type will always be accessed by a component name except in the following contexts: the sequence of expressions in a derived-type value constructor, intrinsic assignment, the data values in namelist input data, and the inclusion of the structure in an input/output list of a formatted data transfer, where it is expanded to this sequence of components. Provided the processor adheres to the defined order in these cases, it is otherwise free to organize the storage of the components for any nonsequence structure in memory as best suited to the particular architecture.

7.5.2.4 Determination of derived types

1 Derived-type definitions with the same type name may appear in different scoping units, in which case they might be independent and describe different derived types or they might describe the same type.

2 Two data entities have the same type if they are declared with reference to the same derived-type definition. Data entities also have the same type if they are declared with reference to different derived-type definitions that specify the same type name, all have the SEQUENCE attribute or all have the BIND attribute, have no components with PRIVATE accessibility, and have components that agree in order, name, and attributes. Otherwise, they are of different derived types. A data entity declared using a type with the SEQUENCE attribute or with the BIND attribute is not of the same type as an entity of a type that has any components that are PRIVATE.
NOTE 7.19
An example of declaring two entities with reference to the same derived-type definition is:

```fortran
TYPE POINT
    REAL X, Y
END TYPE POINT
TYPE (POINT) :: X1
CALL SUB (X1)
...
CONTAINS
    SUBROUTINE SUB (A)
        TYPE (POINT) :: A
    ...
END SUBROUTINE SUB
```

The definition of derived type POINT is known in subroutine SUB by host association. Because the declarations of X1 and A both reference the same derived-type definition, X1 and A have the same type. X1 and A also would have the same type if the derived-type definition were in a module and both SUB and its containing program unit accessed that derived type from the module.

NOTE 7.20
An example of data entities in different scoping units having the same type is:

```fortran
PROGRAM PGM
    TYPE EMPLOYEE
        SEQUENCE
        INTEGER ID_NUMBER
        CHARACTER (50) NAME
    END TYPE EMPLOYEE
    TYPE (EMPLOYEE) PROGRAMMER
    CALL SUB (PROGRAMMER)
    ...
END PROGRAM PGM
SUBROUTINE SUB (POSITION)
    TYPE EMPLOYEE
        SEQUENCE
        INTEGER ID_NUMBER
        CHARACTER (50) NAME
    END TYPE EMPLOYEE
    TYPE (EMPLOYEE) POSITION
    ...
END SUBROUTINE SUB
```

The actual argument PROGRAMMER and the dummy argument POSITION have the same type because they are declared with reference to a derived-type definition with the same name, the SEQUENCE attribute, and components that agree in order, name, and attributes.

Suppose the component name ID_NUMBER was ID_NUM in the subroutine. Because all the component names are not identical to the component names in derived type EMPLOYEE in the main program, the actual argument PROGRAMMER would not be of the same type as the dummy argument POSITION. Thus, the program would not be standard-conforming.

NOTE 7.21
The requirement that the two types have the same name applies to the type-names in the respective derived
type definitions, not to local names introduced via renaming in USE statements.

### 7.5.3 Derived-type parameters

#### 7.5.3.1 Type parameter definition statement

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>R732 type-param-def-stmt is integer-type-spec, type-param-attr-spec ::</td>
</tr>
<tr>
<td>4</td>
<td>type-param-decl-list</td>
</tr>
<tr>
<td>5</td>
<td>R733 type-param-decl is type-param-name [ = scalar-int-constant-expr ]</td>
</tr>
</tbody>
</table>

**C742** (R732) A type-param-name in a type-param-def-stmt in a derived-type-def shall be one of the type-param-names in the derived-type-stmt of that derived-type-def.

**C743** (R732) Each type-param-name in the derived-type-stmt in a derived-type-def shall appear exactly once as a type-param-name in a type-param-def-stmt in that derived-type-def.

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>R734 type-param-attr-spec is KIND</td>
</tr>
<tr>
<td></td>
<td>or LEN</td>
</tr>
</tbody>
</table>

1. The derived type is parameterized if the derived-type-stmt has any type-param-names.

2. Each type parameter is itself of type integer. If its kind selector is omitted, the kind type parameter is default integer.

3. The type-param-attr-spec explicitly specifies whether a type parameter is a kind parameter or a length parameter.

4. If a type-param-decl has a scalar-int-constant-expr, the type parameter has a default value which is specified by the expression. If necessary, the value is converted according to the rules of intrinsic assignment (10.2.1.3) to a value of the same kind as the type parameter.

5. A type parameter may be used as a primary in a specification expression (10.1.11) in the derived-type-def. A kind type parameter may also be used as a primary in a constant expression (10.1.12) in the derived-type-def.

### NOTE 7.22

The following example uses derived-type parameters.

```fortran
TYPE humongous_matrix(k, d)
  INTEGER, KIND :: k = KIND (0.0)
  INTEGER (SELECTED_INT_KIND (12)), LEN :: d
  !-- Specify a potentially nondefault kind for d.
  REAL (k) :: element (d, d)
END TYPE
```

In the following example, dim is declared to be a kind parameter, allowing generic overloading of procedures distinguished only by dim.

```fortran
TYPE general_point(dim)
  INTEGER, KIND :: dim
  REAL :: coordinates(dim)
END TYPE
```

#### 7.5.3.2 Type parameter order

1. Type parameter order is an ordering of the type parameters of a derived type; it is used for derived-type specifiers.
The type parameter order of a nonextended type is the order of the type parameter list in the derived-type definition. The type parameter order of an extended type (7.5.7) consists of the type parameter order of its parent type followed by any additional type parameters in the order of the type parameter list in the derived-type definition.

NOTE 7.23

Given

```fortran
TYPE :: t1 (k1, k2)
  INTEGER, KIND :: k1, k2
  REAL (k1) a (k2)
END TYPE

TYPE, EXTENDS(t1) :: t2 (k3)
  INTEGER, KIND :: k3
  LOGICAL (k3) flag
END TYPE
```

the type parameter order for type t1 is k1 then k2, and the type parameter order for type t2 is k1 then k2 then k3.

7.5.4 Components

7.5.4.1 Component definition statement

R735 component-part is [ component-def-stmt ] ...

R736 component-def-stmt is data-component-def-stmt
  or proc-component-def-stmt

R737 data-component-def-stmt is declaration-type-spec [ [ , component-attr-spec-list ] :: ]
  component-decl-list

R738 component-attr-spec is access-spec
  or ALLOCATABLE
  or CODIMENSION lbracket coarray-spec rbracket
  or CONTIGUOUS
  or DIMENSION ( component-array-spec )
  or POINTER

R739 component-decl is component-name [ ( component-array-spec ) ]
  [ lbracket coarray-spec rbracket ]
  [ * char-length ] [ component-initialization ]

R740 component-array-spec is explicit-shape-spec-list
  or deferred-shape-spec-list

C744 (R737) No component-attr-spec shall appear more than once in a given component-def-stmt.

C745 (R737) If neither the POINTER nor the ALLOCATABLE attribute is specified, the declaration-type-spec in the component-def-stmt shall specify an intrinsic type or a previously defined derived type.

C746 (R737) If the POINTER or ALLOCATABLE attribute is specified, each component-array-spec shall be a deferred-shape-spec-list.

C747 (R737) If a coarray-spec appears, it shall be a deferred-coshape-spec-list and the component shall have the ALLOCATABLE attribute.
C748 (R737) If a coarray-spec appears, the component shall not be of type C_PTR or C_FUNPTR from the intrinsic module ISO_C_BINDING (18.3.3).

C749 A data component whose type has a coarray ultimate component shall be a nonpointer nonallocatable scalar and shall not be a coarray.

C750 (R737) If neither the POINTER nor the ALLOCATABLE attribute is specified, each component-array-spec shall be an explicit-shape-spec-list.

C751 (R740) Each bound in the explicit-shape-spec shall be a specification expression in which there are no references to specification functions or the intrinsic functions ALLOCATED, ASSOCIATED, EXTENDS_TYPE_OF, PRESENT, or SAME_TYPE_AS, every specification inquiry reference is a constant expression, and the value does not depend on the value of a variable.

C752 (R737) A component shall not have both the ALLOCATABLE and POINTER attributes.

C753 (R737) If the CONTIGUOUS attribute is specified, the component shall be an array with the POINTER attribute.

C754 (R739) The * char-length option is permitted only if the component is of type character.

C755 (R736) Each type-param-value within a component-def-stmt shall be a colon or a specification expression in which there are no references to specification functions or the intrinsic functions ALLOCATED, ASSOCIATED, EXTENDS_TYPE_OF, PRESENT, or SAME_TYPE_AS, every specification inquiry reference is a constant expression, and the value does not depend on the value of a variable.

NOTE 7.24
Because a type parameter is not an object, a type-param-value or a bound in an explicit-shape-spec can contain a type-param-name.

R741 proc-component-def-stmt is PROCEDURE ( [ proc-interface ] ) , 

 procedural-component-attr-spec-list :: proc-decl-list

NOTE 7.25
See 15.4.3.6 for definitions of proc-interface and proc-decl.

R742 procedural-component-attr-spec is access-spec 
or NOPASS 
or PASS [ (arg-name) ] 
or POINTER

C756 (R741) The same procedural-component-attr-spec shall not appear more than once in a given proc-component-def-stmt.

C757 (R741) POINTER shall appear in each procedural-component-attr-spec-list.

C758 (R741) If the procedure pointer component has an implicit interface or has no arguments, NOPASS shall be specified.

C759 (R741) If PASS (arg-name) appears, the interface of the procedure pointer component shall have a dummy argument named arg-name.

C760 (R741) PASS and NOPASS shall not both appear in the same procedural-component-attr-spec-list.

The declaration-type-spec in the data-component-def-stmt specifies the type and type parameters of the components in the component-decl-list, except that the character length parameter may be specified or overridden for a component by the appearance of * char-length in its entity-decl. The component-attr-spec-list in the data-component-def-stmt specifies the attributes whose keywords appear for the components in the component-decl-list,
except that the DIMENSION attribute may be specified or overridden for a component by the appearance of a component-array-spec in its component-decl, and the CODIMENSION attribute may be specified or overridden for a component by the appearance of a coarray-spec in its component-decl.

### 7.5.4.2 Array components

A data component is an array if its component-decl contains a component-array-spec or its data-component-def-stmt contains a DIMENSION clause. If the component-decl contains a component-array-spec, it specifies the array rank, and if the array is explicit shape (8.5.8.2), the array bounds; otherwise, the component-array-spec in the DIMENSION clause specifies the array rank, and if the array is explicit shape, the array bounds.

**NOTE 7.26**

An example of a derived type definition with an array component is:

```fortran
TYPE LINE
  REAL, DIMENSION (2, 2) :: COORD !
  ! COORD(:,1) has the value of [X1, Y1]
  ! COORD(:,2) has the value of [X2, Y2]
  REAL :: WIDTH ! Line width in centimeters
  INTEGER :: PATTERN ! 1 for solid, 2 for dash, 3 for dot
END TYPE LINE
```

An example of declaring a variable LINE_SEGMENT to be of the type LINE is:

```fortran
TYPE (LINE) :: LINE_SEGMENT
```

The scalar variable LINE_SEGMENT has a component that is an array. In this case, the array is a subobject of a scalar. The double colon in the definition for COORD is required; the double colon in the definition for WIDTH and PATTERN is optional.

**NOTE 7.27**

An example of a derived type definition with an allocatable component is:

```fortran
TYPE STACK
  INTEGER :: INDEX
  INTEGER, ALLOCATABLE :: CONTENTS (:)
END TYPE STACK
```

For each scalar variable of type STACK, the shape of the component CONTENTS is determined by execution of an ALLOCATE statement or assignment statement, or by argument association.

**NOTE 7.28**

Default initialization of an explicit-shape array component can be specified by a constant expression consisting of an array constructor (7.8), or of a single scalar that becomes the value of each array element.

### 7.5.4.3 Coarray components

A data component is a coarray if its component-decl contains a coarray-spec or its data-component-def-stmt contains a CODIMENSION clause. If the component-decl contains a coarray-spec, it specifies the corank; otherwise, the coarray-spec in the CODIMENSION clause specifies the corank.

**NOTE 7.29**

An example of a derived type definition with a coarray component is:

```fortran
TYPE GRID_TYPE
```

76 ISO/IEC JTC 1/SC 22/WG5/N2118
NOTE 7.29 (cont.)

```fortran
REAL, ALLOCATABLE, CODIMENSION [:, :, :] :: GRID (:, :, :)
END TYPE GRID_TYPE
```

An object of type grid_type cannot be an array, an allocatable object, a coarray, or a pointer.

7.5.4.4 Pointer components

A data component is a pointer (5.4.9) if its component-attr-spec-list contains the POINTER keyword. A procedure pointer component has the POINTER keyword in its proc-component-attr-spec-list.

NOTE 7.30

An example of a derived type definition with a pointer component is:

```fortran
TYPE REFERENCE
  INTEGER :: VOLUME, YEAR, PAGE
  CHARACTER (LEN = 50) :: TITLE
  PROCEDURE (printer_interface), POINTER :: PRINT => NULL()
  CHARACTER, DIMENSION (:), POINTER :: SYNOPSIS
END TYPE REFERENCE
```

Any object of type REFERENCE will have the four nonpointer components VOLUME, YEAR, PAGE, and TITLE, the procedure pointer PRINT, which has an explicit interface the same as printer_interface, plus a pointer to an array of characters holding SYNOPSIS. The size of this target array will be determined by the length of the synopsis. The space for the target could be allocated (9.7.1) or the pointer component could be associated with a target by a pointer assignment statement (10.2.2).

7.5.4.5 The passed-object dummy argument

A passed-object dummy argument is a distinguished dummy argument of a procedure pointer component or type-bound procedure (7.5.5). It affects procedure overriding (7.5.7.3) and argument association (15.5.2.2).

1 If NOPASS is specified, the procedure pointer component or type-bound procedure has no passed-object dummy argument.

2 If neither PASS nor NOPASS is specified or PASS is specified without arg-name, the first dummy argument of a procedure pointer component or type-bound procedure is its passed-object dummy argument.

3 If PASS (arg-name) is specified, the dummy argument named arg-name is the passed-object dummy argument of the procedure pointer component or named type-bound procedure.

C761 The passed-object dummy argument shall be a scalar, nonpointer, nonallocatable dummy data object with the same declared type as the type being defined; all of its length type parameters shall be assumed; it shall be polymorphic (7.3.2.3) if and only if the type being defined is extensible (7.5.7). It shall not have the VALUE attribute.

NOTE 7.31

If a procedure is bound to several types as a type-bound procedure, different dummy arguments might be the passed-object dummy argument in different contexts.

7.5.4.6 Default initialization for components

Default initialization provides a means of automatically initializing pointer components to be disassociated or associated with specific targets, and nonpointer nonallocatable components to have a particular value. Allocatable components are always initialized to unallocated.
A pointer variable or component is data-pointer-initialization compatible with a target if the pointer is type compatible with the target, they have the same rank, all nondeferred type parameters of the pointer have the same values as the corresponding type parameters of the target, and the target is contiguous if the pointer has the CONTIGUOUS attribute.

**R743** component-initialization \( = \) constant-expr

or \( => \) null-init

or \( => \) initial-data-target

**R744** initial-data-target \( = \) designator

(R737) If component-initialization appears, a double-colon separator shall appear before the component-decl-list.

(R737) If component-initialization appears, every type parameter and array bound of the component shall be a colon or constant expression.

(R737) If \( => \) appears in component-initialization, POINTER shall appear in the component-attr-spec-list. If \( = \) appears in component-initialization, neither POINTER nor ALLOCATABLE shall appear in the component-attr-spec-list.

(R743) If initial-data-target appears, component-name shall be data-pointer-initialization compatible with it.

(R744) The designator shall designate a nonallocatable, noncoindexed variable that has the TARGET and SAVE attributes and does not have a vector subscript. Every subscript, section subscript, substring starting point, and substring ending point in designator shall be a constant expression.

3 If null-init appears for a pointer component, that component in any object of the type has an initial association status of disassociated (3.61) or becomes disassociated as specified in 19.5.2.4.

4 If initial-data-target appears for a data pointer component, that component in any object of the type is initially associated with the target or becomes associated with the target as specified in 19.5.2.3.

5 If initial-proc-target (15.4.3.6) appears in proc-decl for a procedure pointer component, that component in any object of the type is initially associated with the target or becomes associated with the target as specified in 19.5.2.3.

6 If constant-expr appears for a nonpointer component, that component in any object of the type is initially defined (19.6.3) or becomes defined as specified in 19.6.5 with the value determined from constant-expr. If necessary, the value is converted according to the rules of intrinsic assignment (10.2.1.3) to a value that agrees in type, type parameters, and shape with the component. If the component is of a type for which default initialization is specified for a component, the default initialization specified by constant-expr overrides the default initialization specified for that component. When one initialization overrides another it is as if only the overriding initialization were specified (see Note 7.33). Explicit initialization in a type declaration statement (8.2) overrides default initialization (see Note 7.32). Unlike explicit initialization, default initialization does not imply that the object has the SAVE attribute.

7 A subcomponent (9.4.2) is default-initialized if the type of the object of which it is a component specifies default initialization for that component, and the subcomponent is not a subobject of an object that is default-initialized or explicitly initialized.

8 A type has default initialization if component-initialization is specified for any direct component of the type. An object has default initialization if it is of a type that has default initialization.

**NOTE 7.32**

It is not required that initialization be specified for each component of a derived type. For example:
NOTE 7.32 (cont.)

```
TYPE DATE
  INTEGER DAY
  CHARACTER (LEN = 5) MONTH
  INTEGER :: YEAR = 2008 ! Partial default initialization
END TYPE DATE
```

In the following example, the default initial value for the YEAR component of TODAY is overridden by explicit initialization in the type declaration statement:

```
TYPE (DATE), PARAMETER :: TODAY = DATE (21, "Feb.", 2009)
```

NOTE 7.33

The default initial value of a component of derived type can be overridden by default initialization specified in the definition of the type. Continuing the example of Note 7.32:

```
TYPE SINGLE_SCORE
  TYPE (DATE) :: PLAY_DAY = TODAY
  INTEGER SCORE
END TYPE SINGLE_SCORE
```

```
TYPE (SINGLE_SCORE), POINTER :: NEXT => NULL ( )
```

```
END TYPE SINGLE_SCORE
```

```
TYPE (SINGLE_SCORE) SETUP
```

```
The PLAY_DAY component of SETUP receives its initial value from TODAY, overriding the initialization for the YEAR component.
```

NOTE 7.34

Arrays of structures can be declared with elements that are partially or totally initialized by default. Continuing the example of Note 7.33:

```
TYPE MEMBER (NAME_LEN)
  INTEGER, LEN :: NAME_LEN
  CHARACTER (LEN = NAME_LEN) :: NAME = '
  INTEGER :: TEAM_NO, HANDICAP = 0
  TYPE (SINGLE_SCORE), POINTER :: HISTORY => NULL ( )
END TYPE MEMBER
```

```
TYPE (MEMBER(9)) LEAGUE (36) ! Array of partially initialized elements
```

```
TYPE (MEMBER(9)) :: ORGANIZER = MEMBER (9) ("I. Manage",1,5,NULL ( ))
```

```
ORGANIZER is explicitly initialized, overriding the default initialization for an object of type MEMBER.
```

```
Allocated objects can also be initialized partially or totally. For example:
```

```
ALLOCATE (ORGANIZER % HISTORY) ! A partially initialized object of type
```

```
! SINGLE_SCORE is created.
```

NOTE 7.35

A pointer component of a derived type can have as its target an object of that derived type. The type definition can specify that in objects declared to be of this type, such a pointer is default initialized to disassociated. For example:

```
TYPE NODE
  INTEGER :: VALUE = 0
  TYPE (NODE), POINTER :: NEXT_NODE => NULL ( )
END TYPE NODE
```

```
ISO/IEC JTC 1/SC 22/WG5/N2118 79
```
NOTE 7.35 (cont.)

END TYPE

A type such as this can be used to construct linked lists of objects of type NODE. Linked lists can also be constructed using allocatable components.

NOTE 7.36

A pointer component of a derived type can be default initialized to have an initial target.

```fortran
TYPE NODE
  INTEGER :: VALUE = 0
  TYPE (NODE), POINTER :: NEXT_NODE => SENTINEL
END TYPE
```

```fortran
TYPE(NODE), SAVE, TARGET :: SENTINEL
```

### 7.5.4.7 Component order

1. **Component order** is an ordering of the nonparent components of a derived type; it is used for intrinsic formatted input/output and structure constructors where component keywords are not used. Parent components are excluded from the component order of an extended type (7.5.7).

2. The component order of a nonextended type is the order of the declarations of the components in the derived-type definition. The component order of an extended type consists of the component order of its parent type followed by any additional components in the order of their declarations in the extended derived-type definition.

NOTE 7.37

Given the same type definitions as in Note 7.23, the component order of type T1 is just A (there is only one component), and the component order of type T2 is A then FLAG. The parent component (T1) does not participate in the component order.

### 7.5.4.8 Component accessibility

1. The default accessibility for the components that are declared in a type’s component-part is private if the type definition contains a private-components-stmt, and public otherwise. The accessibility of a component may be explicitly declared by an access-spec; otherwise its accessibility is the default for the type definition in which it is declared.

2. If a component is private, that component name is accessible only within the module containing the definition, and within its descendants.

NOTE 7.38

Type parameters are not components. They are effectively always public.

NOTE 7.39

The accessibility of the components of a type is independent of the accessibility of the type name. It is possible to have all four combinations: a public type name with a public component, a private type name with a private component, a public type name with a private component, and a private type name with a public component.
NOTE 7.40
An example of a public type with private components is:

```fortran
TYPE, PUBLIC :: POINT
  PRIVATE
  REAL :: X, Y
END TYPE POINT
```

Such a type definition can be accessed by use association; however, the components X and Y are accessible only within the module and its descendants.

NOTE 7.41
The following example illustrates the use of an individual component access-spec to override the default accessibility:

```fortran
TYPE MIXED
  PRIVATE
  INTEGER :: I
  INTEGER, PUBLIC :: J
END TYPE MIXED

TYPE (MIXED) :: M
```

The component M%J is accessible in any scoping unit where M is accessible; M%I is accessible only within the module containing the TYPE MIXED definition, and within its descendants.

### 7.5.5 Type-bound procedures

1. If `=> procedure-name` does not appear in a `type-bound-proc-decl`, it is as though `=> procedure-name` had appeared with a procedure name the same as the binding name. 

R746  `type-bound-procedure-part` is `contains-stmt`

- `[ binding-private-stmt ]`
- `[ type-bound-proc-binding ] ...

R747  `binding-private-stmt` is `PRIVATE`

C768  (R746) A `binding-private-stmt` is permitted only if the type definition is within the specification part of a module.

R748  `type-bound-proc-binding` is `type-bound-procedure-stmt`

- or `type-bound-generic-stmt`
- or `final-procedure-stmt`

R749  `type-bound-procedure-stmt` is `PROCEDURE [ , binding-attr-list ] :: ] type-bound-proc-decl-list`

- or `PROCEDURE (interface-name), binding-attr-list :: binding-name-list`

R750  `type-bound-proc-decl` is `binding-name [ => procedure-name ]`

C769  (R749) If `=> procedure-name` appears in a `type-bound-proc-decl`, the double-colon separator shall appear.

C770  (R750) The `procedure-name` shall be the name of an accessible module procedure or an external procedure that has an explicit interface.

C771  A `binding-name` in a `type-bound-proc-decl` in a derived type definition shall not be the same as any other `binding-name` within that derived type definition.
R751  type-bound-generic-stmt  is  GENERIC [ , access-spec ] :: generic-spec => binding-name-list

C772  (R751) Within the specification-part of a module, each type-bound-generic-stmt shall specify, either implicitly or explicitly, the same accessibility as every other type-bound-generic-stmt with that generic-spec in the same derived type.

C773  (R751) Each binding-name in binding-name-list shall be the name of a specific binding of the type.

C774  A binding-name in a type-bound GENERIC statement shall not specify a specific binding that was inherited or specified previously for the same generic identifier in that derived type definition.

C775  (R751) If generic-spec is not generic-name, each of its specific bindings shall have a passed-object dummy argument (7.5.4.5).

C776  (R751) If generic-spec is OPERATOR ( defined-operator ), the interface of each binding shall be as specified in 15.4.3.4.2.

C777  (R751) If generic-spec is ASSIGNMENT ( = ), the interface of each binding shall be as specified in 15.4.3.4.3.

C778  (R751) If generic-spec is defined-io-generic-spec, the interface of each binding shall be as specified in 12.6.4.8. The type of the dtv argument shall be type-name.

R752  binding-attr  is  access-spec

or  DEFERRED

or  NON_OVERRIDABLE

or  NOPASS

or  PASS [ (arg-name) ]

C779  (R752) The same binding-attr shall not appear more than once in a given binding-attr-list.

C780  (R749) If the interface of the binding has no dummy argument of the type being defined, NOPASS shall appear.

C781  (R749) If PASS (arg-name) appears, the interface of the binding shall have a dummy argument named arg-name.

C782  (R752) PASS and NOPASS shall not both appear in the same binding-attr-list.

C783  (R752) NON_OVERRIDABLE and DEFERRED shall not both appear in the same binding-attr-list.

C784  (R752) DEFERRED shall appear if and only if interface-name appears.

C785  (R749) An overriding binding (7.5.7.3) shall have the DEFERRED attribute only if the binding it overrides is deferred.

C786  (R749) A binding shall not override an inherited binding (7.5.7.2) that has the NON_OVERRIDABLE attribute.

2  A type-bound procedure statement declares one or more specific type-bound procedures. A specific type-bound procedure can have a passed-object dummy argument (7.5.4.5). A type-bound procedure with the DEFERRED attribute is a deferred type-bound procedure. The DEFERRED keyword shall appear only in the definition of an abstract type.

3  A GENERIC statement declares a generic type-bound procedure, which is a type-bound generic interface for its specific type-bound procedures.

4  A binding of a type is a type-bound procedure (specific or generic), a generic type-bound interface, or a final subroutine. These are referred to as specific bindings, generic bindings, and final bindings respectively.
A type-bound procedure may be identified by a binding name in the scope of the type definition. This name is the binding-name for a specific type-bound procedure, and the generic-name for a generic binding whose generic-spec is generic-name. A final binding, or a generic binding whose generic-spec is not generic-name, has no binding name.

The interface of a specific type-bound procedure is that of the procedure specified by procedure-name or the interface specified by interface-name.

NOTE 7.42
An example of a type and a type-bound procedure is:

```
TYPE POINT
    REAL :: X, Y
END TYPE POINT
...
```

and in the module-subprogram-part of the same module:

```
REAL FUNCTION POINT_LENGTH (A, B)
    CLASS (POINT), INTENT (IN) :: A, B
    POINT_LENGTH = SQRT ((A%X - B%X)**2 + (A%Y - B%Y)**2 )
END FUNCTION POINT_LENGTH
```

The same generic-spec may be used in several GENERIC statements within a single derived-type definition. Each additional GENERIC statement with the same generic-spec extends the generic interface.

NOTE 7.43
Unlike the situation with generic procedure names, a generic type-bound procedure name is not permitted to be the same as a specific type-bound procedure name in the same type (19.3).

The default accessibility for the type-bound procedures of a type is private if the type definition contains a binding-private-stmt, and public otherwise. The accessibility of a type-bound procedure may be explicitly declared by an access-spec; otherwise its accessibility is the default for the type definition in which it is declared.

A public type-bound procedure is accessible via any accessible object of the type. A private type-bound procedure is accessible only within the module containing the type definition, and within its descendants.

NOTE 7.44
The accessibility of a type-bound procedure is not affected by a PRIVATE statement in the component-part; the accessibility of a data component is not affected by a PRIVATE statement in the type-bound-procedure-part.

7.5.6 Final subroutines

7.5.6.1 FINAL statement

R753 final-procedure-stmt is FINAL [ :: ] final-subroutine-name-list

C787 (R753) A final-subroutine-name shall be the name of a module procedure with exactly one dummy argument. That argument shall be nonoptional and shall be a noncoarray, nonpointer, nonallocatable, nonpolymorphic variable of the derived type being defined. All length type parameters of the dummy argument shall be assumed. The dummy argument shall not have the INTENT (OUT) or VALUE attribute.
A \textit{final-subroutine-name} shall not be one previously specified as a \textit{final subroutine} for that type.

A \textit{final subroutine} shall not have a dummy argument with the same kind type parameters and rank as the dummy argument of another \textit{final subroutine} of that type.

If a \textit{final subroutine} has an assumed-rank dummy argument, no other \textit{final subroutine} of that type shall have a dummy argument with the same kind type parameters.

The \texttt{FINAL} statement specifies that each procedure it names is a \textit{final subroutine}. A \textit{final subroutine} might be executed when a data entity of that type is finalized (7.5.6.2).

A derived type is \texttt{finalizable} if and only if it has a \textit{final subroutine} or a nonpointer, nonallocatable component of \texttt{finalizable} type. A nonpointer data entity is \texttt{finalizable} if and only if it is of \texttt{finalizable} type. No other entity is \texttt{finalizable}.

\textbf{NOTE 7.45}

\textit{Final subroutines} are effectively always “accessible”. They are called for entity \texttt{finalization} regardless of the accessibility of the type, its other \textit{type-bound procedures}, or the subroutine name itself.

\textbf{NOTE 7.46}

\textit{Final subroutines} are not\texttt{inherited} through type extension and cannot be overridden. The \textit{final subroutines} of the parent type are called after any additional \textit{final subroutines} of an \texttt{extended type} are called.

\section*{7.5.6.2 The finalization process}

Only \texttt{finalizable} entities are finalized. When an entity is \texttt{finalized}, the following steps are carried out in sequence.

1. If the \texttt{dynamic type} of the entity has a \textit{final subroutine} whose dummy argument has the same kind type parameters and rank as the entity being finalized, it is called with the entity as an \texttt{actual argument}. Otherwise, if there is an \texttt{elemental final subroutine} whose dummy argument has the same kind type parameters as the entity being finalized, or a \textit{final subroutine} whose dummy argument is assumed-rank with the same kind type parameters as the entity being finalized, it is called with the entity as an \texttt{actual argument}. Otherwise, no subroutine is called at this point.

2. All \texttt{finalizable} components that appear in the type definition are finalized in a processor-dependent order. If the entity being finalized is an array, each \texttt{finalizable} component of each element of that entity is finalized separately.

3. If the entity is of \texttt{extended type} and the parent type is \texttt{finalizable}, the parent component is finalized.

If several entities are to be finalized as a consequence of an event specified in 7.5.6.3, the order in which they are finalized is processor dependent. During this process, execution of a \textit{final subroutine} for one of these entities shall not reference or define any of the other entities that have already been finalized.

\textbf{NOTE 7.47}

An implementation might need to ensure that when an event causes more than one \texttt{coarray} to be deallocated, they are deallocated in the same order on all \texttt{images} in the current team.

\section*{7.5.6.3 When finalization occurs}

When an \texttt{intrinsic assignment statement} is executed (10.2.1.3), if the variable is not an unallocated allocatable variable, it is finalized after evaluation of \texttt{expr} and before the definition of the variable. If the variable is an allocated allocatable variable, or has an allocated allocatable subobject, that would be deallocated by intrinsic assignment, the finalization occurs before the deallocation.

When a pointer is deallocated its \texttt{target} is finalized. When an \texttt{allocatable} entity is deallocated, it is finalized unless it is the variable in an \texttt{intrinsic assignment statement} or a subobject thereof. If an error condition occurs during deallocation, it is processor dependent whether finalization occurs.
A nonpointer, nonallocatable object that is not a dummy argument or function result is finalized immediately before it would become undefined due to execution of a RETURN or END statement (19.6.6, item (3)).

A nonpointer nonallocatable local variable of a BLOCK construct is finalized immediately before it would become undefined due to termination of the BLOCK construct (19.6.6, item (23)).

If an executable construct references a nonpointer function, the result is finalized after execution of the innermost executable construct containing the reference.

If a specification expression in a scoping unit references a function, the result is finalized before execution of the executable constructs in the scoping unit.

When a procedure is invoked, a nonpointer, nonallocatable, INTENT (OUT) dummy argument of that procedure is finalized before it becomes undefined. The finalization caused by INTENT (OUT) is considered to occur within the invoked procedure; so for elemental procedures, an INTENT (OUT) argument will be finalized only if a scalar or elemental final subroutine is available, regardless of the rank of the actual argument.

If an object is allocated via pointer allocation and later becomes unreachable due to all pointers associated with that object having their pointer association status changed, it is processor dependent whether it is finalized. If it is finalized, it is processor dependent as to when the final subroutines are called.

NOTE 7.48

If finalization is used for storage management, it often needs to be combined with defined assignment.

7.5.6.4 Entities that are not finalized

If image execution is terminated, either by an error (e.g. an allocation failure) or by execution of a stop-stmt, error-stop-stmt, or end-program-stmt, entities existing immediately prior to termination are not finalized.

NOTE 7.49

A nonpointer, nonallocatable object that has the SAVE attribute is never finalized as a direct consequence of the execution of a RETURN or END statement.

7.5.7 Type extension

7.5.7.1 Extensible, extended, and abstract types

A derived type, other than the type C_PTR or C_FUNPTR from the intrinsic module ISO_C_BINDING, that does not have the BIND attribute or the SEQUENCE attribute is an extensible type.

A type with the EXTENDS attribute is an extended type; its parent type is the type named in the EXTENDS type-attr-spec.

The name of the parent type might be a local name introduced via renaming in a USE statement.

An extensible type that does not have the EXTENDS attribute is an extension type of itself only. An extended type is an extension of itself and of all types for which its parent type is an extension.

An abstract type is a type that has the ABSTRACT attribute.

NOTE 7.51

The DEFERRED attribute (7.5.5) defers the implementation of a type-bound procedure to extensions of the type; it can appear only in an abstract type. The dynamic type of an object cannot be abstract; therefore, a deferred type-bound procedure cannot be invoked. An extension of an abstract type need not be abstract if it has no deferred type-bound procedures. A short example of an abstract type is:
NOTE 7.51 (cont.)

```plaintext
TYPE, ABSTRACT :: FILE_HANDLE
CONTAINS
    PROCEDURE(OPEN_FILE), DEFERRED, PASS(HANDLE) :: OPEN
    ...
END TYPE
```

For a more elaborate example see C.2.4.

7.5.7.2 Inheritance

An extended type includes all of the type parameters, all of the components, and the nonoverridden (7.5.7.3) type-bound procedures of its parent type. These are inherited by the extended type from the parent type. They retain all of the attributes that they had in the parent type. Additional type parameters, components, and procedure bindings may be declared in the derived-type definition of the extended type.

NOTE 7.52

Inaccessible components and bindings of the parent type are also inherited, but they remain inaccessible in the extended type. Inaccessible entities occur if the type being extended is accessed via use association and has a private entity.

NOTE 7.53

An extensible derived type is not required to have any components, bindings, or parameters; an extended type is not required to have more components, bindings, or parameters than its parent type.

An extended type has a scalar, nonpointer, nonallocatable, parent component with the type and type parameters of the parent type. The name of this component is the parent type name. If the extended type is defined in a module, the parent component has the accessibility of the parent type in the module in which the parent type was defined. Components of the parent component are inheritance associated (19.5.4) with the corresponding components inherited from the parent type. An ancestor component of a type is the parent component of the type or an ancestor component of the parent component.

If a generic binding specified in a type definition has the same generic-spec as an inherited binding, it extends the generic interface and shall satisfy the requirements specified in 15.4.3.4.5.

NOTE 7.54

A component or type parameter declared in an extended type cannot have the same name as any accessible component or type parameter of its parent type.

NOTE 7.55

Examples:

```plaintext
TYPE POINT ! A base type
    REAL :: X, Y
END TYPE POINT

TYPE, EXTENDS(POINT) :: COLOR_POINT ! An extension of TYPE(POINT)
    ! Components X and Y, and component name POINT, inherited from parent
    INTEGER :: COLOR
END TYPE COLOR_POINT
```
7.5.7.3 Type-bound procedure overriding

1. If a specific type-bound procedure specified in a type definition has the same binding name as an accessible type-bound procedure from the parent type then the binding specified in the type definition overrides the one from the parent type.

2. The overriding and overridden type-bound procedures shall satisfy the following conditions.
   - Either both shall have a passed-object dummy argument or neither shall.
   - If the overridden type-bound procedure is pure then the overriding one shall also be pure.
   - Either both shall be elemental or neither shall.
   - They shall have the same number of dummy arguments.
   - Passed-object dummy arguments, if any, shall correspond by name and position.
   - Dummy arguments that correspond by position shall have the same names and characteristics, except for the type of the passed-object dummy arguments.
   - Either both shall be subroutines or both shall be functions having the same result characteristics (15.3.3).
   - If the overridden type-bound procedure is PUBLIC then the overriding one shall not be PRIVATE.

3. A binding of a type and a binding of an extension of that type correspond if the latter binding is the same binding as the former, overrides a corresponding binding, or is an inherited corresponding binding.

NOTE 7.5.6
The following is an example of procedure overriding, expanding on the example in Note 7.42.

```
TYPE, EXTENDS (POINT) :: POINT_3D
  REAL :: Z
END TYPE POINT_3D

CONTAINS
  PROCEDURE, PASS :: LENGTH => POINT_3D_LENGTH
END TYPE POINT_3D

REAL FUNCTION POINT_3D_LENGTH ( A, B )
CLASS (POINT_3D), INTENT (IN) :: A
CLASS (POINT), INTENT (IN) :: B
SELECT TYPE(B)
  CLASS IS(POINT_3D)
    POINT_3D_LENGTH = SQRT( (A%X-B%X)**2 + (A%Y-B%Y)**2 + (A%Z-B%Z)**2 )
    RETURN
  END SELECT
PRINT *, 'In POINT_3D_LENGTH, dynamic type of argument is incorrect.'
END FUNCTION POINT_3D_LENGTH
```

7.5.8 Derived-type values

1. The component value of
   - a pointer component is its pointer association,
   - an allocatable component is its allocation status and, if it is allocated, its dynamic type and type parameters, bounds and value, and
   - a nonpointer nonallocatable component is its value.

2. The set of values of a particular derived type consists of all possible sequences of the component values of its components.
7.5.9 Derived-type specifier

A derived-type specifier is used in several contexts to specify a particular derived type and type parameters.

R754  derived-type-spec is type-name [ ( type-param-spec-list ) ]
R755  type-param-spec is [ keyword = ] type-param-value

C791 (R754) type-name shall be the name of an accessible derived type.
C792 (R754) type-param-spec-list shall appear only if the type is parameterized.
C793 (R754) There shall be at most one type-param-spec corresponding to each parameter of the type. If a type parameter does not have a default value, there shall be a type-param-spec corresponding to that type parameter.

C794 (R755) The keyword= may be omitted from a type-param-spec only if the keyword= has been omitted from each preceding type-param-spec in the type-param-spec-list.
C795 (R755) Each keyword shall be the name of a parameter of the type.
C796 (R755) An asterisk may be used as a type-param-value in a type-param-spec only in the declaration of a dummy argument or associate name or in the allocation of a dummy argument.

2 Type parameter values that do not have type parameter keywords specified correspond to type parameters in type parameter order (7.5.3.2). If a type parameter keyword appears, the value corresponds to the type parameter named by the keyword. If necessary, the value is converted according to the rules of intrinsic assignment (10.2.1.3) to a value of the same kind as the type parameter.

3 The value of a type parameter for which no type-param-value has been specified is its default value.

7.5.10 Construction of derived-type values

A derived-type definition implicitly defines a corresponding structure constructor that allows construction of scalar values of that derived type. The type and type parameters of a constructed value are specified by a derived type specifier.

R756 structure-constructor is derived-type-spec ( [ component-spec-list ] )
R757 component-spec is [ keyword = ] component-data-source
R758 component-data-source is expr
or data-target
or proc-target

C797 (R756) The derived-type-spec shall not specify an abstract type (7.5.7).
C798 (R756) At most one component-spec shall be provided for a component.
C799 (R756) If a component-spec is provided for an ancestor component, a component-spec shall not be provided for any component that is inheritance associated with a subcomponent of that ancestor component.
C7100 (R756) A component-spec shall be provided for a nonallocatable component unless it has default initialization or is inheritance associated with a subcomponent of another component for which a component-spec is provided.
C7101 (R757) The keyword= may be omitted from a component-spec only if the keyword= has been omitted from each preceding component-spec in the constructor.
C7102 (R757) Each keyword shall be the name of a component of the type.
C7103  (R756) The type name and all components of the type for which a component-spec appears shall be accessible in the scoping unit containing the structure constructor.

C7104  (R756) If derived-type-spec is a type name that is the same as a generic name, the component-spec-list shall not be a valid actual-arg-spec-list for a function reference that is resolvable as a generic reference to that name (15.5.5.2).

C7105  (R758) A data-target shall correspond to a data pointer component; a proc-target shall correspond to a procedure pointer component.

C7106  (R758) A data-target shall have the same rank as its corresponding component.

NOTE 7.57

The form 'name(...)’ is interpreted as a generic function-reference if possible; it is interpreted as a structure-constructor only if it cannot be interpreted as a generic function-reference.

In the absence of a component keyword, each component-data-source is assigned to the corresponding component in component order (7.5.4.7). If a component keyword appears, the expr is assigned to the component named by the keyword. For a nonpointer component, the declared type and type parameters of the component and expr shall conform in the same way as for a variable and expr in an intrinsic assignment statement (10.2.1.2). If necessary, each value of intrinsic type is converted according to the rules of intrinsic assignment (10.2.1.3) to a value that agrees in type and type parameters with the corresponding component of the derived type. For a nonpointer nonallocatable component, the shape of the expression shall conform with the shape of the component.

NOTE 7.58

Because no parent components appear in the defined component ordering, a value for a parent component can be specified only with a component keyword. Examples of equivalent values using types defined in Note 7.55:

! Create values with components x = 1.0, y = 2.0, color = 3.
TYPE(POINT) :: PV = POINT(1.0, 2.0)  ! Assume components of TYPE(POINT)
          ! are accessible here.
...
COLOR_POINT( point=point(1,2), color=3)  ! Value for parent component
COLOR_POINT( point=PV, color=3)         ! Available even if TYPE(point)
          ! has private components
COLOR_POINT( 1, 2, 3)                   ! All components of TYPE(point)
          ! need to be accessible.

A structure constructor shall not appear before the referenced type is defined.

NOTE 7.59

This example illustrates a derived-type constant expression using a derived type defined in Note 7.15:

PERSON (21, ’JOHN SMITH’)

This could also be written as

PERSON (NAME = ’JOHN SMITH’, AGE = 21)
NOTE 7.60

An example constructor using the derived type GENERAL_POINT defined in Note 7.22 is

general_point(dim=3) ( [ 1., 2., 3. ] )

For a pointer component, the corresponding component-data-source shall be an allowable data-target or proc-target for such a pointer in a pointer assignment statement (10.2.2). If the component data source is a pointer, the association of the component is that of the pointer; otherwise, the component is pointer associated with the component data source.

NOTE 7.61

For example, if the variable TEXT were declared (8.2) to be

CHARACTER, DIMENSION (1:400), TARGET :: TEXT

and BIBLIO were declared using the derived-type definition REFERENCE in Note 7.30

TYPE (REFERENCE) :: BIBLIO

the statement

BIBLIO = REFERENCE (1, 1987, 1, "This is the title of the referenced & paper", SYNOPSIS=TEXT)

is valid and associates the pointer component SYNOPSIS of the object BIBLIO with the target object TEXT. The keyword SYNOPSIS is required because the fifth component of the type REFERENCE is a procedure pointer component, not a data pointer component of type character. It is not necessary to specify a proc-target for the procedure pointer component because it has default initialization.

If a component of a derived type is allocatable, the corresponding constructor expression shall be a reference to the intrinsic function NULL with no arguments, an allocatable entity of the same rank, or shall evaluate to an entity of the same rank. If the expression is a reference to the intrinsic function NULL, the corresponding component of the constructor has a status of unallocated.

If the component is allocatable and the expression is an allocatable entity, the corresponding component of the constructor has the same allocation status as that allocatable entity. If it is allocated, it has the same bounds; if a length parameter of the component is deferred, its value is the same as the corresponding parameter of the expression. If the component is polymorphic, it has the same dynamic type and value; otherwise, it has the value converted, if necessary, to the declared type of the component.

If the component is allocatable and the expression is not an allocatable entity, the component has an allocation status of allocated and the same bounds as the expression; if a length parameter of the component is deferred, its value is the same as the corresponding parameter of the expression. If the component is polymorphic, it has the same dynamic type and value; otherwise, it has the value converted, if necessary, to the declared type of the component.

7.5.11 Derived-type operations and assignment

Intrinsic assignment of derived-type entities is described in 10.2.1. This document does not specify any intrinsic operations on derived-type entities. Any operation on derived-type entities or defined assignment (10.2.1.4) for derived-type entities shall be defined explicitly by a function or a subroutine, and a generic interface (7.5.5, 15.4.3.2).
7.6 Enumerations and enumerators

An enumeration is a set of enumerators. An enumerator is a named integer constant. An enumeration definition specifies the enumeration and its set of enumerators of the corresponding integer kind.

R759 enum-def is enum-def-stmt
   enumerator-def-stmt
   [ enumerator-def-stmt ] ...
   end-enum-stmt

R760 enum-def-stmt is ENUM, BIND(C)

R761 enumerator-def-stmt is ENUMERATOR [ :: ] enumerator-list

R762 enumerator is named-constant [ = scalar-int-constant-expr ]

R763 end-enum-stmt is END ENUM

C7107 (R761) If = appears in an enumerator, a double-colon separator shall appear before the enumerator-list.

For an enumeration, the kind is selected such that an integer type with that kind is interoperable (18.3.2) with the corresponding C enumeration type. The corresponding C enumeration type is the type that would be declared by a C enumeration specifier (6.7.2.2 of ISO/IEC 9899:2011) that specified C enumeration constants with the same values as those specified by the enum-def, in the same order as specified by the enum-def.

The companion processor (5.5.7) shall be one that uses the same representation for the types declared by all C enumeration specifiers that specify the same values in the same order.

NOTE 7.62
If a companion processor uses an unsigned type to represent a given enumeration type, the Fortran processor will use the signed integer type of the same width for the enumeration, even though some of the values of the C enumerators might not be representable in this signed integer type. The types of any such enumerators will be interoperable with the type declared in the C enumeration.

NOTE 7.63
ISO/IEC 9899:2011 guarantees the enumeration constants fit in a C int (6.7.2.2 of ISO/IEC 9899:2011). Therefore, the Fortran processor can evaluate all enumerator values using the integer type with kind parameter C_INT, and then determine the kind parameter of the integer type that is interoperable with the corresponding C enumerated type.

NOTE 7.64
ISO/IEC 9899:2011 specifies that two enumeration types are compatible only if they specify enumeration constants with the same names and same values in the same order. This document further requires that a C processor that is to be a companion processor of a Fortran processor use the same representation for two enumeration types if they both specify enumeration constants with the same values in the same order, even if the names are different.

An enumerator is treated as if it were explicitly declared with the PARAMETER attribute. The enumerator is defined in accordance with the rules of intrinsic assignment (10.2) with the value determined as follows.

(1) If scalar-int-constant-expr is specified, the value of the enumerator is the result of scalar-int-constant-expr.

(2) If scalar-int-constant-expr is not specified and the enumerator is the first enumerator in enum-def, the enumerator has the value 0.

(3) If scalar-int-constant-expr is not specified and the enumerator is not the first enumerator in enum-def, its value is the result of adding 1 to the value of the enumerator that immediately precedes it.
Example of an enumeration definition:

```
ENUM, BIND(C)
    ENUMERATOR :: RED = 4, BLUE = 9
    ENUMERATOR YELLOW
END ENUM
```

The kind type parameter for this enumeration is processor dependent, but the processor is required to select a kind sufficient to represent the values 4, 9, and 10, which are the values of its enumerators. The following declaration might be equivalent to the above enumeration definition.

```
INTEGER (SELECTED_INT_KIND (2)), PARAMETER :: RED = 4, BLUE = 9, YELLOW = 10
```

An entity of the same kind type parameter value can be declared using the intrinsic function KIND with one of the enumerators as its argument, for example

```
INTEGER (KIND (RED)) :: X
```

There is no difference in the effect of declaring the enumerators in multiple ENUMERATOR statements or in a single ENUMERATOR statement. The order in which the enumerators in an enumeration definition are declared is significant, but the number of ENUMERATOR statements is not.

### 7.7 Binary, octal, and hexadecimal literal constants

A binary, octal, or hexadecimal constant (boz-literal-constant) is a sequence of digits that represents an ordered sequence of bits. Such a constant has no type.

- **boz-literal-constant**
  - is \textit{binary-constant} or \textit{octal-constant} or \textit{hex-constant}

- **binary-constant**
  - is \texttt{B} \texttt{'} \texttt{digit} [...] \texttt{'}
  - or \texttt{B} " \texttt{digit} [...] "

- **octal-constant**
  - is \texttt{O} \texttt{'} \texttt{digit} [...] \texttt{'}
  - or \texttt{O} " \texttt{digit} [...] "

- **hex-constant**
  - is \texttt{Z} \texttt{'} \texttt{hex-digit} [...] \texttt{'}
  - or \texttt{Z} " \texttt{hex-digit} [...] "

- **hex-digit**
  - is \texttt{digit}
  - or A
  - or B
  - or C
  - or D
  - or E
  - or F

The \textit{hex-digits} A through F represent the numbers ten through fifteen, respectively; they may be represented...
by their lower-case equivalents. Each digit of a boz-literal-constant represents a sequence of bits, according to its numerical interpretation, using the model of 16.3, with z equal to one for binary constants, three for octal constants or four for hexadecimal constants. A boz-literal-constant represents a sequence of bits that consists of the concatenation of the sequences of bits represented by its digits, in the order the digits are specified. The positions of bits in the sequence are numbered from right to left, with the position of the rightmost bit being zero. The length of a sequence of bits is the number of bits in the sequence. The processor shall allow the position of the leftmost nonzero bit to be at least $z - 1$, where $z$ is the maximum value that could result from invoking the intrinsic function STORAGE_SIZE (16.9.184) with an argument that is a real or integer scalar of any kind supported by the processor.

C7110 (R764) A boz-literal-constant shall appear only as a data-stmt-constant in a DATA statement, or where explicitly allowed in subclause 16.9 as an actual argument of an intrinsic procedure.

### 7.8 Construction of array values

An array constructor constructs a rank-one array value from a sequence of scalar values, array values, and implied DO loops.

R769 array-constructor is (/ ac-spec /)

or lbracket ac-spec rbracket

R770 ac-spec is type-spec ::

or [type-spec ::] ac-value-list

R771 lbracket is [

R772 rbracket is ]

R773 ac-value is expr

or ac-implied-do

R774 ac-implied-do is ( ac-value-list , ac-implied-do-control )

R775 ac-implied-do-control is [ integer-type-spec :: ] ac-do-variable = scalar-int-expr ,

■ scalar-int-expr [, scalar-int-expr ]

R776 ac-do-variable is do-variable

C7111 (R770) If type-spec is omitted, each ac-value expression in the array-constructor shall have the same declared type and kind type parameters.

C7112 (R770) If type-spec specifies an intrinsic type, each ac-value expression in the array-constructor shall be of an intrinsic type that is in type conformance with a variable of type type-spec as specified in Table 10.8.

C7113 (R770) If type-spec specifies a derived type, the declared type of each ac-value expression in the array-constructor shall be that derived type and shall have the same kind type parameter values as specified by type-spec.

C7114 (R773) An ac-value shall not be unlimited polymorphic.

C7115 (R773) The declared type of an ac-value shall not be abstract.

C7116 (R774) The ac-do-variable of an ac-implied-do that is in another ac-implied-do shall not appear as the ac-do-variable of the containing ac-implied-do.

2 If type-spec is omitted, corresponding length type parameters of the declared type of each ac-value expression shall have the same value; in this case, the declared type and type parameters of the array constructor are those of the ac-value expressions.
If `type-spec` appears, it specifies the declared type and type parameters of the array constructor. Each `ac-value` expression in the `array-constructor` shall be compatible with intrinsic assignment to a variable of this type and type parameters. Each value is converted to the type and type parameters of the `array-constructor` in accordance with the rules of intrinsic assignment (10.2.1.3).

The dynamic type of an array constructor is the same as its declared type.

The character length of an `ac-value` in an `ac-implied-do` whose iteration count is zero shall not depend on the value of the `ac-do-variable` and shall not depend on the value of an expression that is not a constant expression.

If an `ac-value` is a scalar expression, its value specifies an element of the array constructor. If an `ac-value` is an array expression, the values of the elements of the expression, in array element order (9.5.3.2), specify the corresponding sequence of elements of the array constructor. If an `ac-value` is an `ac-implied-do`, it is expanded to form a sequence of elements under the control of the `ac-do-variable`, as in the DO construct (11.1.7.4).

For an `ac-implied-do`, the loop initialization and execution is the same as for a DO construct.

An empty sequence forms a zero-sized array.

**NOTE 7.67**

A one-dimensional array can be reshaped into any allowable array shape using the intrinsic function RESHAPE (16.9.163). An example is:

```
X = (/ 3.2, 4.01, 6.5 /)
Y = RESHAPE (SOURCE = [ 2.0, [ 4.5, 4.5 ], X ], SHAPE = [ 3, 2 ])
```

This results in Y having the $3 \times 2$ array of values:

```
2.0  3.2
4.5  4.01
4.5  6.5
```

**NOTE 7.68**

Examples of array constructors containing an implied DO are:

```
(/ (I, I = 1, 1075) /)
```

and

```
[ 3.6, (3.6 / I, I = 1, N) ]
```

**NOTE 7.69**

Using the type definition for PERSON in Note 7.15, an example of the construction of a derived-type array value is:

```
[ PERSON (40, 'SMITH'), PERSON (20, 'JONES') ]
```

**NOTE 7.70**

Using the type definition for LINE in Note 7.26, an example of the construction of a derived-type scalar value with a rank-2 array component is:

```
LINE (RESHAPE ( [ 0.0, 0.0, 1.0, 2.0 ], [ 2, 2 ] ), 0.1, 1)
```

The intrinsic function RESHAPE is used to construct a value that represents a solid line from (0, 0) to (1, 2) of width 0.1 centimeters.
NOTE 7.71

Examples of zero-size array constructors are:

\[
[\text{INTEGER} :: ]
[ ( I, I = 1, 0 ) ]
\]

NOTE 7.72

An example of an array constructor that specifies a length type parameter:

\[
[ \text{CHARACTER(LEN=7)} :: 'Takata', 'Tanaka', 'Hayashi' ]
\]

In this constructor, without the type specification, it would have been necessary to specify all of the constants with the same character length.
8 Attribute declarations and specifications

8.1 Attributes of procedures and data objects

Every data object has a type and rank and may have type parameters and other properties that determine the uses of the object. Collectively, these properties are the attributes of the object. The declared type of a named data object is either specified explicitly in a type declaration statement or determined implicitly by the first letter of its name (8.7). All of its attributes may be specified in a type declaration statement or individually in separate specification statements.

A function has a type and rank and may have type parameters and other attributes that determine the uses of the function. The type, rank, and type parameters are the same as those of the function result.

A subroutine does not have a type, rank, or type parameters, but may have other attributes that determine the uses of the subroutine.

8.2 Type declaration statement

R801 type-declaration-stmt is declaration-type-spec [] attr-spec ] ... :: ] entity-decl-list

The type declaration statement specifies the declared type of the entities in the entity declaration list. The type and type parameters are those specified by declaration-type-spec, except that the character length type parameter may be overridden for an entity by the appearance of * char-length in its entity-decl.

R802 attr-spec is access-spec

or ALLOCATABLE

or ASYNCHRONOUS

or CODIMENSION lbracket coarray-spec rbracket

or CONTIGUOUS

or DIMENSION ( array-spec )

or EXTERNAL

or INTENT ( intent-spec )

or INTRINSIC

or language-binding-spec

or OPTIONAL

or PARAMETER

or POINTER

or PROTECTED

or SAVE

or TARGET

or VALUE

or VOLATILE

C801 (R801) The same attr-spec shall not appear more than once in a given type-declaration-stmt.

C802 (R801) If a language-binding-spec with a NAME= specifier appears, the entity-decl-list shall consist of a single entity-decl.

C803 (R801) If a language-binding-spec is specified, the entity-decl-list shall not contain any procedure names.

The type declaration statement also specifies the attributes whose keywords appear in the attr-spec, except that
the DIMENSION attribute may be specified or overridden for an entity by the appearance of array-spec in its entity-decl, and the CODIMENSION attribute may be specified or overridden for an entity by the appearance of coarray-spec in its entity-decl.

R803  entity-decl is  object-name [ ( array-spec ) ]
      [ [ lbracket coarray-spec rbracket ]]
      [ [ * char-length ] [ initialization ]]
      or  function-name [ * char-length ]

C804  (R803) If the entity is not of type character, * char-length shall not appear.

C805  (R801) If initialization appears, a double-colon separator shall appear before the entity-decl-list.

C806  (R801) If the PARAMETER keyword appears, initialization shall appear in each entity-decl.

C807  (R803) An initialization shall not appear if object-name is a dummy argument, a function result, an object in a named common block unless the type declaration is in a block data program unit, an object in blank common, an allocatable variable, or an automatic data object.

C808  (R803) The function-name shall be the name of an external function, an intrinsic function, a dummy function, a procedure pointer, or a statement function.

R804  object-name is  name

C809  (R804) The object-name shall be the name of a data object.

R805  initialization is  = constant-expr
      or  => null-init
      or  => initial-data-target

R806  null-init is  function-reference

C810  (R803) If => appears in initialization, the entity shall have the POINTER attribute. If = appears in initialization, the entity shall not have the POINTER attribute.

C811  (R803) If initial-data-target appears, object-name shall be data-pointer-initialization compatible with it (7.5.4.6).

C812  (R806) The function-reference shall be a reference to the intrinsic function NULL with no arguments.

3 A name that identifies a specific intrinsic function has a type as specified in 16.8. An explicit type declaration statement is not required; however, it is permitted. Specifying a type for a generic intrinsic function name in a type declaration statement has no effect.

4 If initialization appears for a nonpointer entity,
   • its type and type parameters shall conform as specified for intrinsic assignment (10.2.1.2);
   • if the entity has implied shape, the rank of initialization shall be the same as the rank of the entity;
   • if the entity does not have implied shape, initialization shall either be scalar or have the same shape as the entity.

NOTE 8.1
Examples of type declaration statements:

    REAL A (10)
    LOGICAL, DIMENSION (5, 5) :: MASK1, MASK2
    COMPLEX :: CUBE_ROOT = (-0.5, 0.866)
    INTEGER, PARAMETER :: SHORT = SELECTED_INT_KIND (4)
    INTEGER (SHORT) K  ! Range at least -9999 to 9999.
NOTE 8.1 (cont.)

| REAL (KIND (0.0D0)) B1 |
| REAL (KIND = 2) B2 |
| COMPLEX (KIND = KIND (0.0D0)) :: C |
| CHARACTER (LEN = 10, KIND = 2) TEXT2 |
| CHARACTER CHAR, STRING *20 |
| TYPE (PERSON) :: CHAIRMAN |
| TYPE (NODE), POINTER :: HEAD => NULL ( ) |
| TYPE (humongous_matrix (k=8, d=1000)) :: MAT |

(The last line above uses a type definition from Note 7.22.)

8.3 Automatic data objects

1 An automatic data object is a non-dummy data object with a type parameter or array bound that depends on the value of a specification-expr that is not a constant expression.

C813 An automatic data object shall not have the SAVE attribute.

2 If a type parameter in a declaration-type-spec or in a char-length in an entity-decl for a local variable of a subprogram or BLOCK construct is defined by an expression that is not a constant expression, the type parameter value is established on entry to a procedure defined by the subprogram, or on execution of the BLOCK statement, and is not affected by any redefinition or undefinition of the variables in the expression during execution of the procedure or BLOCK construct.

8.4 Initialization

1 The appearance of initialization in an entity-decl for an entity without the PARAMETER attribute specifies that the entity is a variable with explicit initialization. Explicit initialization alternatively may be specified in a DATA statement unless the variable is of a derived type for which default initialization is specified. If initialization is = constant-expr, the variable is initially defined with the value specified by the constant-expr; if necessary, the value is converted according to the rules of intrinsic assignment (10.2.1.3) to a value that agrees in type, type parameters, and shape with the variable. A variable, or part of a variable, shall not be explicitly initialized more than once in a program. If the variable is an array, it shall have its shape specified in either the type declaration statement or a previous attribute specification statement in the same scoping unit.

2 If null-init appears, the initial association status of the object is disassociated. If initial-data-target appears, the object is initially associated with the target.

3 Explicit initialization of a variable that is not in a common block implies the SAVE attribute, which may be confirmed by explicit specification.

8.5 Attributes

8.5.1 Attribute specification

1 An attribute may be explicitly specified by an attr-spec in a type declaration statement or by an attribute specification statement (8.6). The following constraints apply to attributes.

C814 An entity shall not be explicitly given any attribute more than once in a scoping unit.

C815 An array-spec for a nonallocatable nonpointer function result shall be an explicit-shape-spec-list.
### 8.5.2 Accessibility attribute

1. The accessibility attribute specifies the accessibility of an entity via a particular identifier.

```plaintext
R807 access-spec is PUBLIC
or PRIVATE
```

2. An `access-spec` shall appear only in the `specification-part` of a module.

3. An `access-spec` in a type declaration statement specifies the accessibility of the names of all the entities declared by that statement. An `access-spec` in a `derived-type-stmt` specifies the accessibility of the derived type name. Accessibility can also be specified by an `access-stmt`.

4. An identifier that is specified in a module or is accessible in a module by use association has either the `PUBLIC` attribute or `PRIVATE` attribute. An identifier whose accessibility is not explicitly specified has default accessibility (8.6.1).

5. The default accessibility attribute for a module is `PUBLIC` unless it has been changed by a `PRIVATE` statement. Only an identifier that has the `PUBLIC` attribute in that module is available to be accessed from that module by use association.

#### NOTE 8.2

An identifier can only be accessed by use association if it has the `PUBLIC` attribute in the module from which it is accessed. It can nonetheless have the `PRIVATE` attribute in a module in which it is accessed by use association, and therefore not be available by use association from that module.

#### NOTE 8.3

An example of an accessibility specification is:

```plaintext
REAL, PRIVATE :: X, Y, Z
```

### 8.5.3 ALLOCATABLE attribute

1. A variable with the `ALLOCATABLE` attribute is a variable for which space is allocated during execution.

### 8.5.4 ASYNCHRONOUS attribute

1. An entity with the `ASYNCHRONOUS` attribute is a variable that may be subject to asynchronous input/output or asynchronous communication.

2. The base object of a variable shall have the `ASYNCHRONOUS` attribute in a scoping unit if

   - the variable is a dummy argument or appears in an executable statement or specification expression in that scoping unit, and
   - any statement of the scoping unit is executed while the variable is a pending input/output storage sequence affector (12.6.2.5) or a pending communication affector (18.10.4).

3. Use of a variable in an asynchronous data transfer statement can imply the `ASYNCHRONOUS` attribute; see subclause 12.6.2.5.

4. An object with the `ASYNCHRONOUS` attribute may be associated with an object that does not have the `ASYNCHRONOUS` attribute, including by use (14.2.2) or host association (19.5.1.4). If an object that is not a local variable of a `BLOCK` construct is specified to have the `ASYNCHRONOUS` attribute in the `specification-part` of the construct, the object has the attribute within the construct even if it does not have the attribute outside the construct. If an object has the `ASYNCHRONOUS` attribute, then all of its subobjects also have the `ASYNCHRONOUS` attribute.
NOTE 8.4

The **ASYNCHRONOUS** attribute specifies the variables that might be associated with a pending input/output storage sequence (the actual memory locations on which asynchronous input/output is being performed) while the **scoping unit** is in execution. This information could be used by the compiler to disable certain code motion optimizations.

### 8.5.5 BIND attribute for data entities

1 The **BIND** attribute for a variable or **common block** specifies that it is capable of interoperating with a C variable whose name has external linkage (18.9).

R808 `language-binding-spec  is  BIND (C [, NAME = scalar-default-char-constant-expr ])

C817 An entity with the **BIND** attribute shall be a **common block**, variable, type, or procedure.

C818 A variable with the **BIND** attribute shall be declared in the specification part of a module.

C819 A variable with the **BIND** attribute shall be interoperable (18.3).

C820 Each variable of a **common block** with the **BIND** attribute shall be interoperable.

2 If the value of the `scalar-default-char-constant-expr` after discarding leading and trailing blanks has nonzero length, it shall be valid as an identifier on the companion processor.

NOTE 8.5

ISO/IEC 9899:2011 provides a facility for creating C identifiers whose characters are not restricted to the C basic character set. Such a C identifier is referred to as a universal character name (6.4.3 of ISO/IEC 9899:2011). The name of such a C identifier might include characters that are not part of the representation method used by the processor for default character. If so, the C entity cannot be referenced from Fortran.

3 The **BIND** attribute for a **common block** implies the **SAVE** attribute, which may be confirmed by explicit specification.

### 8.5.6 CODIMENSION attribute

**8.5.6.1 General**

1 The **CODIMENSION** attribute specifies that an entity is a **coarray**. The **coarray-spec** specifies its corank or corank and cobounds.

R809 `coarray-spec  is  deferred-coshape-spec-list

or  explicit-coshape-spec`

C821 The sum of the **rank** and **corank** of an entity shall not exceed fifteen.

C822 A **coarray** shall be a component or a variable that is not a function result.

C823 A **coarray** shall not be of type **C_PTR** or **C_FUNPTR** from the intrinsic module **ISO_C_BINDING** (18.3.3).

C824 An entity whose type has a **coarray ultimate component** shall be a nonpointer nonallocatable scalar, shall not be a **coarray**, and shall not be a function result.

C825 A **coarray** or an object with a **coarray ultimate component** shall be a **dummy argument** or have the **ALLOCATABLE** or **SAVE** attribute.
NOTE 8.6
A coarray is permitted to be of a derived type with pointer or allocatable components. The target of such a pointer component is always on the same image as the pointer.

NOTE 8.7
This requirement for the SAVE attribute has the effect that automatic coarrays are not permitted; for example, the coarray WORK in the following code fragment is not valid.

SUBROUTINE SOLVE3(N,A,B)
INTEGER :: N
REAL :: A(N)[*], B(N)
REAL :: WORK(N)[*] ! Not permitted

If this were permitted, it would require an implicit synchronization on entry to the procedure.

Explicit-shape coarrays that are declared in a subprogram and are not dummy arguments are required to have the SAVE attribute because otherwise they might be implemented as if they were automatic coarrays.

NOTE 8.8
Examples of CODIMENSION attribute specifications are:

| REAL W(100,100)[0:2,∗] ! Explicit-shape coarray |
| REAL, CODIMENSION[*] :: X ! Scalar coarray |
| REAL, CODIMENSION[3,∗] :: Y(:) ! Assumed-shape coarray |
| REAL, CODIMENSION[::],ALLOCATABLE :: Z(:,:) ! Allocatable coarray |

8.5.6.2 Allocatable coarray

A coarray with the ALLOCATABLE attribute has a specified corank, but its cobounds are determined by allocation or argument association.

R810 deferred-coshape-spec is :
C826 A coarray with the ALLOCATABLE attribute shall have a coarray-spec that is a deferred-coshape-spec-list.

2 The corank of an allocatable coarray is equal to the number of colons in its deferred-coshape-spec-list.
3 The cobounds of an unallocated allocatable coarray are undefined. No part of such a coarray shall be referenced or defined; however, the coarray may appear as an argument to an intrinsic inquiry function as specified in 16.1.
4 The cobounds of an allocated allocatable coarray are those specified when the coarray is allocated.
5 The cobounds of an allocatable coarray are unaffected by any subsequent redefinition or undefinition of the variables on which the cobounds’ expressions depend.

8.5.6.3 Explicit-coshape coarray

An explicit-coshape coarray is a named coarray that has its corank and cobounds declared by an explicit-coshape-spec.

R811 explicit-coshape-spec is [ [ lower-cobound : ] upper-cobound, ]... ■
C827 A nonallocatable coarray shall have a coarray-spec that is an explicit-coshape-spec.
2 The corank is equal to one plus the number of upper-cobounds.
R812  lower-cobound  is  specification-expr  
R813  upper-cobound  is  specification-expr  

C828 (R811) A lower-cobound or upper-cobound that is not a constant expression shall appear only in a sub-program, BLOCK construct, or interface body.

3 If an explicit-coshape coarray is a local variable of a subprogram or BLOCK construct and has cobounds that are not constant expressions, the cobounds are determined on entry to a procedure defined by the subprogram, or on execution of the BLOCK statement, by evaluating the cobounds expressions. The cobounds of such a coarray are unaffected by the redefinition or undefinition of any variable during execution of the procedure or BLOCK construct.

4 The values of each lower-cobound and upper-cobound determine the cobounds of the coarray along a particular codimension. The cosubscript range of the coarray in that codimension is the set of integer values between and including the lower and upper cobounds. If the lower cobound is omitted, the default value is 1. The upper cobound shall not be less than the lower cobound.

8.5.7 CONTIGUOUS attribute

C829 An entity with the CONTIGUOUS attribute shall be an array pointer, an assumed-shape array, or an assumed-rank dummy data object.

1 The CONTIGUOUS attribute specifies that an assumed-shape array is contiguous, that an array pointer can only be pointer associated with a contiguous target, or that an assumed-rank dummy data object is contiguous.

2 An object is contiguous if it is

(1) an object with the CONTIGUOUS attribute,  
(2) a nonpointer whole array that is not assumed-shape,  
(3) an assumed-shape array that is argument associated with an array that is contiguous,  
(4) an assumed-rank dummy data object whose effective argument is contiguous,  
(5) an array allocated by an ALLOCATE statement,  
(6) a pointer associated with a contiguous target, or  
(7) a nonzero-sized array section (9.5.3) provided that

(a) its base object is contiguous,  
(b) it does not have a vector subscript,  
(c) the array element ordering of the elements of the section is the same as the array element ordering of those elements of the base object,  
(d) in the array element ordering of the base object, every element of the base object that is not an element of the section either precedes every element of the section or follows every element of the section,  
(e) if the array is of type character and a substring-range appears, the substring-range specifies all of the characters of the parent-string (9.4.1),  
(f) only its final part-ref has nonzero rank, and  
(g) it is not the real or imaginary part (9.4.4) of an array of type complex.

3 An object is not contiguous if it is an array subobject, and

• the object has two or more elements,  
• the elements of the object in array element order are not consecutive in the elements of the base object,  
• the object is not of type character with length zero, and  
• the object is not of a derived type that has no ultimate components other than zero-sized arrays and characters with length zero.
It is processor dependent whether any other object is contiguous.

**NOTE 8.9**

If a derived type has only one component that is not zero-sized, it is processor dependent whether a structure component of a contiguous array of that type is contiguous. That is, the derived type might contain padding on some processors.

**NOTE 8.10**

The **CONTIGUOUS** attribute makes it easier for a processor to enable optimizations that depend on the memory layout of the object occupying a contiguous block of memory. Examples of **CONTIGUOUS** attribute specifications are:

```fortran
REAL, POINTER, CONTIGUOUS :: SPTR(:)
REAL, CONTIGUOUS, DIMENSION(:,:) :: D
```

### 8.5.8 DIMENSION attribute

#### 8.5.8.1 General

The **DIMENSION** attribute specifies that an entity is assumed-rank or an array. An assumed-rank dummy data object has the rank, shape, and size of its effective argument; otherwise, the rank or rank and shape is specified by its `array-spec`.

- **R814** `dimension-spec` is `DIMENSION ( array-spec )`
- **R815** `array-spec` is `explicit-shape-spec-list`
- **R816** `lower-bound` is `specification-expr`
- **R817** `upper-bound` is `specification-expr`

**NOTE 8.11**

The maximum rank of an entity is fifteen minus the corank.

**NOTE 8.12**

Examples of **DIMENSION** attribute specifications are:

```fortran
SUBROUTINE EX (N, A, B)
  REAL, DIMENSION (N, 10) :: W ! Automatic explicit-shape array
  REAL A (:), B (0:) ! Assumed-shape arrays
  REAL, POINTER :: D (:, :) ! Array pointer
  REAL, DIMENSION (:), POINTER :: P ! Array pointer
  REAL, ALLOCATABLE, DIMENSION (:) :: E ! Allocatable array
  REAL, PARAMETER :: V(0:*) = [0.1, 1.1] ! Implied-shape array
ENDSUBROUTINE EX
```

#### 8.5.8.2 Explicit-shape array

- **R816** `explicit-shape-spec` is `[ lower-bound : ] upper-bound`
- **R817** `lower-bound` is `specification-expr`
- **R818** `upper-bound` is `specification-expr`
An explicit-shape array is an array whose shape is explicitly declared by an *explicit-shape-spec-list*. The rank is equal to the number of *explicit-shape-specs*.

An explicit-shape array that is a named local variable of a subprogram or BLOCK construct may have bounds that are not constant expressions. The bounds, and hence shape, are determined on entry to a procedure defined by the subprogram, or on execution of the BLOCK statement, by evaluating the bounds’ expressions. The bounds of such an array are unaffected by the redefinition or undefinition of any variable during execution of the procedure or BLOCK construct.

The values of each *lower-bound* and *upper-bound* determine the bounds of the array along a particular dimension and hence the extent of the array in that dimension. If *lower-bound* appears it specifies the lower bound; otherwise the lower bound is 1. The value of a lower bound or an upper bound may be positive, negative, or zero. The subscript range of the array in that dimension is the set of integer values between and including the lower and upper bounds, provided the upper bound is not less than the lower bound. If the upper bound is less than the lower bound, the range is empty, the extent in that dimension is zero, and the array is of zero size.

**8.5.8.3 Assumed-shape array**

An assumed-shape array is a nonallocatable nonpointer dummy argument array that takes its shape from its effective argument.

The rank is equal to the number of colons in the *assumed-shape-spec-list*.

The extent of a dimension of an assumed-shape array dummy argument is the extent of the corresponding dimension of its effective argument. If the lower bound value is \( d \) and the extent of the corresponding dimension of its effective argument is \( s \), then the value of the upper bound is \( s + d - 1 \). If *lower-bound* appears it specifies the lower bound; otherwise the lower bound is 1.

**8.5.8.4 Deferred-shape array**

A deferred-shape array is an allocatable array or an array pointer. (An allocatable array has the ALLOCATABLE attribute; an array pointer has the POINTER attribute.)

The rank is equal to the number of colons in the *deferred-shape-spec-list*.

The size, bounds, and shape of an unallocated allocatable array or a disassociated array pointer are undefined. No part of such an array shall be referenced or defined; however, the array may appear as an argument to an intrinsic inquiry function as specified in 16.1.

The bounds of each dimension of an allocated allocatable array are those specified when the array is allocated or, if it is a dummy argument, when it is argument associated with an allocated effective argument.

The bounds of each dimension of an associated array pointer, and hence its shape, may be specified

- in an ALLOCATE statement (9.7.1) when the target is allocated,
- by pointer assignment (10.2.2), or
- if it is a dummy argument, by argument association with a nonpointer actual argument or an associated pointer effective argument.
The bounds of an array pointer or allocatable array are unaffected by any subsequent redefinition or undefined
of variables on which the bounds’ expressions depend.

8.5.8.5 Assumed-size array

An assumed-size array is a dummy argument array whose size is assumed from that of its effective argument, or
the associate name of a RANK (*) block in a SELECT RANK construct. The rank and extents may differ for
the effective and dummy arguments; only the size of the effective argument is assumed by the dummy argument.
A dummy argument is declared to be an assumed-size array by an assumed-size-spec or an implied-shape-or-
assumed-size-spec.

The size of an assumed-size array is determined as follows.
• If the effective argument associated with the assumed-size dummy array is an array of any type other than
default character, the size is that of the effective argument.
• If the actual argument corresponding to the assumed-size dummy array is an array element of any type
other than default character with a subscript order value of \( r \) (9.5.3.2) in an array of size \( x \), the size of the
dummy array is \( x - r + 1 \).
• If the actual argument is a default character array, default character array element, or a default character
array element substring (9.4.1), and if it begins at character storage unit \( t \) of an array with \( c \) character
storage units, the size of the dummy array is \( \text{MAX} \left( \text{INT} \left( \left( c - t + 1 \right)/e \right), 0 \right) \), where \( e \) is the length of an
element in the dummy character array.
• If the actual argument is a default character scalar that is not an array element or array element substring
designator, the size of the dummy array is \( \text{MAX} \left( \text{INT} \left( l/e \right), 0 \right) \), where \( e \) is the length of an element in the
dummy character array and \( l \) is the length of the actual argument.

The rank is equal to one plus the number of explicit-shape-specs.

An assumed-size array has no upper bound in its last dimension and therefore has no extent in its last dimension
and no shape. An assumed-size array shall not appear in a context that requires its shape.

If a list of explicit-shape-specs appears, it specifies the bounds of the first rank−1 dimensions. If lower-bound
appears it specifies the lower bound of the last dimension; otherwise that lower bound is 1. An assumed-size
array may be subscripted or sectioned (9.5.3.3). The upper bound shall not be omitted from a subscript triplet
in the last dimension.

If an assumed-size array has bounds that are not constant expressions, the bounds are determined on entry to
the procedure. The bounds of such an array are unaffected by the redefinition or undefined of any variable
during execution of the procedure.

8.5.8.6 Implied-shape array

An implied-shape array is a named constant that takes its shape from the constant-expr in its declaration. A
named constant is declared to be an implied-shape array with an array-spec that is an implied-shape-or-assumed-

ISO/IEC JTC 1/SC 22/WG5/N2118
size-spec or an implied-shape-spec.

R824 implied-shape-spec is assumed-impl-spec, assumed-impl-spec-list

C835 An implied-shape array shall be a named constant.

R825 implied-shape-spec is ..

C836 An assumed-rank entity is a dummy data object whose rank is assumed from its effective argument, or the associate name of a RANK DEFAULT block in a SELECT RANK construct; this rank can be zero. An assumed-rank entity is declared with an array-spec that is an assumed-rank-spec.

C837 An assumed-rank variable name shall not appear in a designator or expression except as an actual argument that corresponds to a dummy argument that is assumed-rank, the argument of the function C_LOC or C_SIZEOF from the intrinsic module ISO_C_BINDING (18.2), the first dummy argument of an intrinsic inquiry function, or the selector of a SELECT RANK statement.

C838 If an assumed-size or nonallocatable nonpointer assumed-rank array is an actual argument that corresponds to a dummy argument that is an INTENT (OUT) assumed-rank array, it shall not be polymorphic, finalizable, of a type with an allocatable ultimate component, or of a type for which default initialization is specified.

8.5.9 EXTERNAL attribute

C839 An entity shall not have both the EXTERNAL attribute and the INTRINSIC attribute.

C840 In an external subprogram, the EXTERNAL attribute shall not be specified for a procedure defined by the subprogram.

C841 In an interface body, the EXTERNAL attribute shall not be specified for the procedure declared by the interface body.

2 If an external procedure or dummy procedure is used as an actual argument or is the target of a procedure pointer assignment, it shall be declared to have the EXTERNAL attribute.

3 A procedure that has both the EXTERNAL and POINTER attributes is a procedure pointer.

NOTE 8.13
The EXTERNAL attribute can be specified in a type declaration statement, by an interface body (15.4.3.2), by an EXTERNAL statement (15.4.3.5), or by a procedure declaration statement (15.4.3.6).

8.5.10 INTENT attribute

C842 The INTENT attribute specifies the intended use of a dummy argument. An INTENT (IN) dummy argument is suitable for receiving data from the invoking scoping unit, an INTENT (OUT) dummy argument is suitable
for returning data to the invoking scoping unit, and an INTENT (INOUT) dummy argument is suitable for use
both to receive data from and to return data to the invoking scoping unit.

R826  intent-spec is IN
      or  OUT
      or  INOUT

C842  An entity with the INTENT attribute shall be a dummy data object or a dummy procedure pointer.
C843  (R826) A nonpointer object with the INTENT (IN) attribute shall not appear in a variable definition
context (19.6.7).
C844  A pointer with the INTENT (IN) attribute shall not appear in a pointer association context (19.6.8).
C845  An INTENT (OUT) dummy argument of a nonintrinsic procedure shall not be an allocatable coarray or
      have a subobject that is an allocatable coarray.
C846  An entity with the INTENT (OUT) attribute shall not be of, or have a subcomponent of, type EVENT_TYPE
      (16.10.2.10) or LOCK_TYPE (16.10.2.19) from the intrinsic module ISO_FORTRAN_ENV.

The INTENT (IN) attribute for a nonpointer dummy argument specifies that it shall neither be defined nor
become undefined during the invocation and execution of the procedure. The INTENT (IN) attribute for a
pointer dummy argument specifies that during the invocation and execution of the procedure its association shall
not be changed except that it may become undefined if the target is deallocated other than through the pointer
(19.5.2.5).

The INTENT (OUT) attribute for a nonpointer dummy argument specifies that the dummy argument becomes
undefined on invocation of the procedure, except for any subcomponents that are default-initialized (7.5.4.6). Any
actual argument that corresponds to such a dummy argument shall be definable. The INTENT (OUT) attribute
for a pointer dummy argument specifies that on invocation of the procedure the pointer association status of
the dummy argument becomes undefined. Any actual argument that corresponds to such a pointer dummy shall
be a pointer variable or a procedure pointer that is not the result of a function reference. Any undefined or
definition implied by association of an actual argument with an INTENT (OUT) dummy argument shall not
affect any other entity within the statement that invokes the procedure.

The INTENT attribute for an allocatable dummy argument applies to both the allocation status and the
definition status. An actual argument that corresponds to an INTENT (OUT) allocatable dummy argument
is deallocated on procedure invocation (9.7.3.2). To avoid this deallocation for coarrays, INTENT (OUT) is
not allowed for a dummy argument that is an allocatable coarray or has a subobject that is an allocatable
coarray.

If no INTENT attribute is specified for a dummy argument, its use is subject to the limitations of its effective
argument (15.5.2).

If an object has an INTENT attribute, then all of its subobjects have the same INTENT attribute.

An example of INTENT specification is:

SUBROUTINE MOVE (FROM, TO)
  USE PERSON_MODULE

NOTE 8.14
NOTE 8.15 (cont.)

| TYPE (PERSON), INTENT (IN) :: FROM |
| TYPE (PERSON), INTENT (OUT) :: TO |

NOTE 8.16

If a dummy argument is a derived-type object with a pointer component, then the pointer as a pointer is a subobject of the dummy argument, but the target of the pointer is not. Therefore, the restrictions on subobjects of the dummy argument apply to the pointer in contexts where it is used as a pointer, but not in contexts where it is dereferenced to indicate its target. For example, if X is a dummy argument of derived type with an integer pointer component P, and X is INTENT (IN), then the statement

X%P => NEW_TARGET

is prohibited, but

X%P = 0

is allowed (provided that X%P is associated with a definable target).

Similarly, the INTENT restrictions on pointer dummy arguments apply only to the association of the dummy argument; they do not restrict the operations allowed on its target.

NOTE 8.17

Argument intent specifications serve several purposes in addition to documenting the intended use of dummy arguments. A processor can check whether an INTENT (IN) dummy argument is used in a way that could redefine it. A slightly more sophisticated processor could check to see whether an INTENT (OUT) dummy argument could possibly be referenced before it is defined. If the procedure’s interface is explicit, the processor can also verify that actual arguments corresponding to INTENT (OUT) or INTENT (INOUT) dummy arguments are definable. A more sophisticated processor could use this information to optimize the translation of the referencing scoping unit by taking advantage of the fact that actual arguments corresponding to INTENT (IN) dummy arguments will not be changed and that any prior value of an actual argument corresponding to an INTENT (OUT) dummy argument will not be referenced and could thus be discarded.

INTENT (OUT) means that the value of the argument after invoking the procedure is entirely the result of executing that procedure. If an argument might not be redefined and it is desired to have the argument retain its value in that case, INTENT (OUT) cannot be used because it would cause the argument to become undefined; however, INTENT (INOUT) can be used, even if there is no explicit reference to the value of the dummy argument.

INTENT (INOUT) is not equivalent to omitting the INTENT attribute. The actual argument corresponding to an INTENT (INOUT) dummy argument is always required to be definable, while an actual argument corresponding to a dummy argument without an INTENT attribute need be definable only if the dummy argument is actually redefined.

8.5.11 INTRINSIC attribute

1 The INTRINSIC attribute specifies that the entity is an intrinsic procedure. The procedure name may be a generic name (16.7), a specific name (16.8), or both.

2 If the specific name of an intrinsic procedure (16.8) is used as an actual argument, the name shall be explicitly specified to have the INTRINSIC attribute. Note that a specific intrinsic procedure listed in Table 16.3 is not permitted to be used as an actual argument (C1534).

3 C847 If the generic name of an intrinsic procedure is explicitly declared to have the INTRINSIC attribute,
and it is also the generic name of one or more generic interfaces (15.4.3.2) accessible in the same scoping unit, the procedures in the interfaces and the generic intrinsic procedure shall all be functions or all be subroutines.

8.5.12 OPTIONAL attribute

1 The OPTIONAL attribute specifies that the dummy argument need not have a corresponding actual argument in a reference to the procedure (15.5.2.12).

C848 An entity with the OPTIONAL attribute shall be a dummy argument.

NOTE 8.18

The intrinsic function PRESENT (16.9.152) can be used to determine whether an optional dummy argument has a corresponding actual argument.

8.5.13 PARAMETER attribute

1 The PARAMETER attribute specifies that an entity is a named constant. The entity has the value specified by its constant-expr, converted, if necessary, to the type, type parameters and shape of the entity.

C849 An entity with the PARAMETER attribute shall not be a variable, a coarray, or a procedure.

C850 An expression that specifies a length type parameter or array bound of a named constant shall be a constant expression.

2 A named constant shall not be referenced unless it has been defined previously; it may be defined previously in the same statement.

NOTE 8.19

Examples of declarations with a PARAMETER attribute are:

```
REAL, PARAMETER :: ONE = 1.0, Y = 4.1 / 3.0
INTEGER, DIMENSION (3), PARAMETER :: ORDER = (/ 1, 2, 3 /)
TYPE(NODE), PARAMETER :: DEFAULT = NODE(0, NULL (^))
```

8.5.14 POINTER attribute

1 Entities with the POINTER attribute can be associated with different data objects or procedures during execution of a program. A pointer is either a data pointer or a procedure pointer.

C851 An entity with the POINTER attribute shall not have the ALLOCATABLE, INTRINSIC, or TARGET attribute, and shall not be a coarray.

C852 A procedure with the POINTER attribute shall have the EXTERNAL attribute.

2 A data pointer shall not be referenced unless it is pointer associated with a target object that is defined. A data pointer shall not be defined unless it is pointer associated with a target object that is definable.

3 If a data pointer is associated, the values of its deferred type parameters are the same as the values of the corresponding type parameters of its target.

4 A procedure pointer shall not be referenced unless it is pointer associated with a target procedure.

NOTE 8.20

Examples of POINTER attribute specifications are:

```
TYPE (NODE), POINTER :: CURRENT, TAIL
```
NOTE 8.20 (cont.)

REAL, DIMENSION (:, :), POINTER :: IN, OUT, SWAP

8.5.15 PROTECTED attribute

1 The PROTECTED attribute imposes limitations on the usage of module entities.

C853 The PROTECTED attribute shall be specified only in the specification part of a module.

C854 An entity with the PROTECTED attribute shall be a procedure pointer or variable.

C855 An entity with the PROTECTED attribute shall not be in a common block.

C856 A nonpointer object that has the PROTECTED attribute and is accessed by use association shall not appear in a variable definition context (19.6.7) or as the data-target or proc-target in a pointer-assignment-stmt.

C857 A pointer that has the PROTECTED attribute and is accessed by use association shall not appear in a pointer association context (19.6.8).

2 Other than within the module in which an entity is given the PROTECTED attribute, or within any of its descendants,

- if it is a nonpointer object, it is not definable, and
- if it is a pointer, its association status shall not be changed except that it may become undefined if its target is deallocated other than through the pointer (19.5.2.5), or if its target becomes undefined by completing execution of a BLOCK construct or by execution of a RETURN or END statement.

3 If an object has the PROTECTED attribute, all of its subobjects have the PROTECTED attribute.

NOTE 8.21

An example of the PROTECTED attribute:

```
MODULE temperature
  REAL, PROTECTED :: temp_c, temp_f
CONTAINS
  SUBROUTINE set_temperature_c(c)
    REAL, INTENT(IN) :: c
    temp_c = c
    temp_f = temp_c*(9.0/5.0) + 32
  END SUBROUTINE
END MODULE
```

The PROTECTED attribute ensures that the variables temp_c and temp_f cannot be modified other than via the set_temperature_c procedure, thus keeping them consistent with each other.

8.5.16 SAVE attribute

1 The SAVE attribute specifies that a local variable of a program unit or subprogram retains its association status, allocation status, definition status, and value after execution of a RETURN or END statement unless it is a pointer and its target becomes undefined (19.5.2.5(6)). If it is a local variable of a subprogram it is shared by all instances (15.6.2.4) of the subprogram.

2 The SAVE attribute specifies that a local variable of a BLOCK construct retains its association status, allocation status, definition status, and value after termination of the construct unless it is a pointer and its target becomes undefined (19.5.2.5(7)). If the BLOCK construct is within a subprogram the variable is shared by all instances (15.6.2.4) of the subprogram.
Giving a common block the SAVE attribute confers the attribute on all entities in the common block.

An entity with the SAVE attribute shall be a common block, variable, or procedure pointer.

The SAVE attribute shall not be specified for a dummy argument, a function result, an automatic data object, or an object that is in a common block.

A variable, common block, or procedure pointer declared in the scoping unit of a main program, module, or submodule implicitly has the SAVE attribute, which may be confirmed by explicit specification. If a common block has the SAVE attribute in any other kind of scoping unit, it shall have the SAVE attribute in every scoping unit that is not of a main program, module, or submodule.

8.5.17 TARGET attribute

The TARGET attribute specifies that a data object may have a pointer associated with it. An object without the TARGET attribute shall not have a pointer associated with it.

An entity with the TARGET attribute shall be a variable.

An entity with the TARGET attribute shall not have the POINTER attribute.

If an object has the TARGET attribute, then all of its nonpointer subobjects also have the TARGET attribute.

NOTE 8.22

In addition to variables explicitly declared to have the TARGET attribute, the objects created by allocation of pointers have the TARGET attribute.

Examples of TARGET attribute specifications are:

```
TYPE (NODE), TARGET :: HEAD
REAL, DIMENSION (1000, 1000), TARGET :: A, B
```

Every object designator that starts from an object with the TARGET attribute will have either the TARGET or POINTER attribute. If pointers are involved, the designator might not necessarily be a subobject of the original object, but because a pointer can point only to an entity with the TARGET attribute, there is no way to end up at a nonpointer that does not have the TARGET attribute.

8.5.18 VALUE attribute

The VALUE attribute specifies a type of argument association for a dummy argument.

An entity with the VALUE attribute shall be a dummy data object. It shall not be an assumed-size array, a coarray, or a variable with a coarray ultimate component.

An entity with the VALUE attribute shall not have the ALLOCATABLE, INTENT (INOUT), INTENT (OUT), POINTER, or VOLATILE attributes.

A dummy argument of a procedure with the BIND attribute shall not have both the OPTIONAL and VALUE attributes.

8.5.19 VOLATILE attribute

The VOLATILE attribute specifies that an object may be referenced, defined, or become undefined, by means not specified by the program. A pointer with the VOLATILE attribute may additionally have its association...
status, dynamic type and type parameters, and array bounds changed by means not specified by the program.

An allocatable object with the VOLATILE attribute may additionally have its allocation status, dynamic type and type parameters, and array bounds changed by means not specified by the program.

C865 An entity with the VOLATILE attribute shall be a variable that is not an INTENT (IN) dummy argument.

C866 The VOLATILE attribute shall not be specified for a coarray, or a variable with a coarray ultimate component, that is accessed by use (14.2.2) or host (19.5.1.4) association.

C867 Within a BLOCK construct (11.1.4), the VOLATILE attribute shall not be specified for a coarray, or a variable with a coarray ultimate component, that is not a construct entity (19.4) of that construct.

A noncoarray object that has the VOLATILE attribute may be associated with an object that does not have the VOLATILE attribute, including by use (14.2.2) or host association (19.5.1.4). If an object that is not a local variable of a BLOCK construct is specified to have the VOLATILE attribute in the specification-part of the construct, the object has the attribute within the construct even if it does not have the attribute outside the construct. The relationship between coarrays, the VOLATILE attribute, and argument association is described in 15.5.2.8. The relationship between between coarrays, the VOLATILE attribute, and pointer association is described in 10.2.2.3.

A pointer should have the VOLATILE attribute if its target has the VOLATILE attribute. If, by means not specified by the program, the target is referenced, defined, or becomes undefined, the pointer shall have the VOLATILE attribute. All members of an EQUIVALENCE group should have the VOLATILE attribute if any member has the VOLATILE attribute.

If an object has the VOLATILE attribute, then all of its subobjects also have the VOLATILE attribute.

The Fortran processor should use the most recent definition of a volatile object each time its value is required. When a volatile object is defined by means of Fortran, it should make that definition available to the non-Fortran parts of the program as soon as possible.

### 8.6 Attribute specification statements

#### 8.6.1 Accessibility statement

An access-stmt with an access-id-list specifies the accessibility attribute, PUBLIC or PRIVATE, of each access-id in the list that is not a module name. An access-stmt without an access-id list specifies the default accessibility of the identifiers of entities declared in the module, and of entities accessed from a module whose name does not appear in any access-stmt in the module. If an identifier is accessed from another module and also declared locally, it has the default accessibility of a locally declared identifier. The statement

PUBLIC

<table>
<thead>
<tr>
<th>Accessibility statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>access-stmt</td>
<td>Access to module is specified by access-spec [ :: ] access-id-list</td>
</tr>
<tr>
<td>access-id</td>
<td>Access to module is specified by access-name</td>
</tr>
<tr>
<td>or generic-spec</td>
<td></td>
</tr>
</tbody>
</table>

ISO/IEC JTC 1/SC 22/WG5/N2118 113
specifies a default of public accessibility. The statement

PRIVATE

specifies a default of private accessibility. If no such statement appears in a module, the default is public
accessibility.

If an identifier is accessed by use association and not declared in the module, and the name of every module
from which it is accessed appears in an access-stmt in the scoping unit, its default accessibility is PRIVATE if
the access-spec in every such access-stmt is PRIVATE, or PUBLIC if the access-spec in any such access-stmt is
PUBLIC.

NOTE 8.25

Examples of accessibility statements are:

MODULE EX
PRIVATE
PUBLIC :: A, B, C, ASSIGNMENT (=), OPERATOR (+)

NOTE 8.26

The following is an example of using an accessibility statement on a module name.

MODULE m2
USE m1
! We want to use the types and procedures in m1, but we only want to
! re-export m_type and our own procedures.
PRIVATE m1
PUBLIC m_type
... definitions for our own entities and module procedures.
END MODULE

8.6.2 ALLOCATABLE statement

The ALLOCATABLE statement specifies the ALLOCATABLE attribute (8.5.3) for a list of objects.

NOTE 8.27

An example of an ALLOCATABLE statement is:

REAL A, B (:), SCALAR
ALLOCATABLE :: A (:, :), B, SCALAR

8.6.3 ASYNCHRONOUS statement

The ASYNCHRONOUS statement specifies the ASYNCHRONOUS attribute (8.5.4) for a list of objects.

8.6.4 BIND statement
(R832) If the language-binding-spec has a NAME= specifier, the bind-entity-list shall consist of a single bind-entity.

1 The BIND statement specifies the BIND attribute for a list of variables and common blocks.

8.6.5 CODIMENSION statement

R834 codimension-stmt is CODIMENSION [ :: ] codimension-decl-list
R835 codimension-decl is coarray-name lbracket coarray-spec rbracket

1 The CODIMENSION statement specifies the CODIMENSION attribute (8.5.6) for a list of objects.

NOTE 8.28
An example of a CODIMENSION statement is:

CODIMENSION a[*], b[3,*], c[:]

8.6.6 CONTIGUOUS statement

R836 contiguous-stmt is CONTIGUOUS [ :: ] object-name-list

1 The CONTIGUOUS statement specifies the CONTIGUOUS attribute (8.5.7) for a list of objects.

8.6.7 DATA statement

R837 data-stmt is DATA data-stmt-set [ [ , ] data-stmt-set ] ...

1 The DATA statement specifies explicit initialization (8.4).
2 If a nonpointer variable has default initialization, it shall not appear in a data-stmt-object-list.
3 A variable that appears in a DATA statement and has not been typed previously may appear in a subsequent type declaration only if that declaration confirms the implicit typing. An array name, array section, or array element that appears in a DATA statement shall have had its array properties established by a previous specification statement.
4 Except for variables in named common blocks, a named variable has the SAVE attribute if any part of it is initialized in a DATA statement, and this may be confirmed by explicit specification.

R838 data-stmt-set is data-stmt-object-list / data-stmt-value-list /
R839 data-stmt-object is variable
or data-implied-do
R840 data-implied-do is ( data-i-do-object-list , [ integer-type-spec :: ] data-i-do-variable =
■ scalar-int-constant-expr ,
■ scalar-int-constant-expr
■ [ , scalar-int-constant-expr ] )
R841 data-i-do-object is array-element
or scalar-structure-component
or data-implied-do
R842 data-i-do-variable is do-variable

C873 A data-stmt-object or data-i-do-object shall not be a coindexed variable.
A data-stmt-object that is a variable shall be a designator. Each subscript, section subscript, substring starting point, and substring ending point in the variable shall be a constant expression.

A variable whose designator appears as a data-stmt-object or a data-i-do-object shall not be a dummy argument, accessed by use or host association, in a named common block unless the DATA statement is in a block data program unit, in blank common, a function name, a function result name, an automatic data object, or an allocatable variable.

A data-i-do-object or a variable that appears as a data-stmt-object shall not be an object designator in which a pointer appears other than as the entire rightmost part-ref.

The array-element shall be a variable.

The scalar-structure-component shall be a variable.

The scalar-structure-component shall contain at least one part-ref that contains a subscript-list.

In an array-element or scalar-structure-component that is a data-i-do-object, any subscript shall be a constant expression, and any primary within that subscript that is a data-i-do-variable shall be a DO variable of this data-implied-do or of a containing data-implied-do.

The data-stmt-value-list is expanded to form a sequence of pointers and scalar variables, referred to as “sequence of variables” in subsequent text. A nonpointer array whose unqualified name appears as a data-stmt-object or data-i-do-object is equivalent to a complete sequence of its array elements in array element order (9.5.3.2). An array section is equivalent to the sequence of its array elements in array element order. A data-implied-do is expanded to form a sequence of array elements and structure components, under the control of the data-i-do-variable, as in the DO construct (11.1.7.4).

The data-stmt-value-list is expanded to form a sequence of data-stmt-constants. A data-stmt-repeat indicates the
number of times the following \textit{data-stmt-constant} is to be included in the sequence; omission of a \textit{data-stmt-repeat} has the effect of a repeat factor of 1.

7 A zero-sized array or a \textit{data-implied-do} with an iteration count of zero contributes no variables to the expanded sequence of variables, but a zero-length scalar character variable does contribute a variable to the expanded sequence. A \textit{data-stmt-constant} with a repeat factor of zero contributes no \textit{data-stmt-constants} to the expanded sequence of scalar \textit{data-stmt-constants}.

8 The expanded sequences of variables and \textit{data-stmt-constants} are in one-to-one correspondence. Each \textit{data-stmt-constant} specifies the initial value, initial data target, or \textit{null-init} for the corresponding variable. The lengths of the two expanded sequences shall be the same.

9 A \textit{data-stmt-constant} shall be \textit{null-init} or \textit{initial-data-target} if and only if the corresponding \textit{data-stmt-object} has the \texttt{POINTER} attribute. If \textit{data-stmt-constant} is \textit{null-init}, the initial association status of the corresponding data statement object is disassociated. If \textit{data-stmt-constant} is \textit{initial-data-target} the corresponding data statement object shall be data-pointer-initialization compatible (7.5.4.6) with the initial data target; the data statement object is initially associated with the target.

A \textit{data-stmt-constant} other than \texttt{boz-literal-constant}, \textit{null-init}, or \textit{initial-data-target} shall be compatible with its corresponding variable according to the rules of intrinsic assignment (10.2.1.2). The variable is initially defined with the value specified by the \textit{data-stmt-constant}; if necessary, the value is converted according to the rules of intrinsic assignment (10.2.1.3) to a value that agrees in type, type parameters, and shape with the variable.

11 If a \textit{data-stmt-constant} is a \texttt{boz-literal-constant}, the corresponding variable shall be of type integer. The \texttt{boz-literal-constant} is treated as if it were converted by the intrinsic function \texttt{INT} (16.9.100) to type integer with the kind type parameter of the variable.

\textbf{NOTE 8.29}

Examples of DATA statements are:

\begin{verbatim}
CHARACTER (LEN = 10) NAME
INTEGER, DIMENSION (0:9) :: MILES
REAL, DIMENSION (100, 100) :: SKEW
TYPE (NODE), POINTER :: HEAD_OF_LIST
TYPE (PERSON) MYNAME, YOURNAME
DATA NAME / 'JOHN DOE' /, MILES / 10 * 0 /
DATA ((SKEW (K, J), J = 1, K), K = 1, 100) / 5050 * 0.0 /
DATA ((SKEW (K, J), J = K + 1, 100), K = 1, 99) / 4950 * 1.0 /
DATA HEAD_OF_LIST / NULL() /
DATA MYNAME / PERSON (21, 'JOHN SMITH') /
DATA YOURNAME % AGE, YOURNAME % NAME / 35, 'FRED BROWN' /
\end{verbatim}

The character variable \texttt{NAME} is initialized with the value \texttt{JOHN DOE} with padding on the right because the length of the constant is less than the length of the variable. All ten elements of the integer array \texttt{MILES} are initialized to zero. The two-dimensional array \texttt{SKEW} is initialized so that the lower triangle of \texttt{SKEW} is zero and the strict upper triangle is one. The structures \texttt{MYNAME} and \texttt{YOURNAME} are declared using the derived type \texttt{PERSON} from Note 7.15. The pointer \texttt{HEAD\_OF\_LIST} is declared using the derived type \texttt{NODE} from Note 7.35; it is initially disassociated. \texttt{MYNAME} is initialized by a structure constructor. \texttt{YOURNAME} is initialized by supplying a separate value for each component.

\section*{8.6.8 DIMENSION statement}

\begin{verbatim}
R848 dimension-stmt is DIMENSION [ :: ] array-name ( array-spec )
\[ [ , array-name ( array-spec ) ] ... \]
\end{verbatim}

The DIMENSION statement specifies the \textit{DIMENSION} attribute (8.5.8) for a list of objects.
NOTE 8.30
An example of a DIMENSION statement is:

```
DIMENSION A (10), B (10, 70), C (:)
```

8.6.9 INTENT statement

```
R849 intent-stmt is INTENT ( intent-spec ) [ :: ] dummy-arg-name-list
```

1 The INTENT statement specifies the INTENT attribute (8.5.10) for the dummy arguments in the list.

NOTE 8.31
An example of an INTENT statement is:

```
SUBROUTINE EX (A, B)
INTENT (INOUT) :: A, B
```

8.6.10 OPTIONAL statement

```
R850 optional-stmt is OPTIONAL [ :: ] dummy-arg-name-list
```

1 The OPTIONAL statement specifies the OPTIONAL attribute (8.5.12) for the dummy arguments in the list.

NOTE 8.32
An example of an OPTIONAL statement is:

```
SUBROUTINE EX (A, B)
OPTIONAL :: B
```

8.6.11 PARAMETER statement

```
R851 parameter-stmt is PARAMETER ( named-constant-def-list )
R852 named-constant-def is named-constant = constant-expr
```

1 The PARAMETER statement specifies the PARAMETER attribute (8.5.13) and the values for the named constants in the list.

2 If a named constant is defined by a PARAMETER statement, it shall not be subsequently declared to have a type or type parameter value that differs from the type and type parameters it would have if declared implicitly (8.7). A named array constant defined by a PARAMETER statement shall have its rank specified in a prior specification statement.

3 The constant expression that corresponds to a named constant shall have type and type parameters that conform with the named constant as specified for intrinsic assignment (10.2.1.2). If the named constant has implied shape, the expression shall have the same rank as the named constant; otherwise, the expression shall either be scalar or have the same shape as the named constant.

4 The value of each named constant is that specified by the corresponding constant expression; if necessary, the value is converted according to the rules of intrinsic assignment (10.2.1.3) to a value that agrees in type, type parameters, and shape with the named constant.

NOTE 8.33
An example of a PARAMETER statement is:

```
PARAMETER (MODULUS = MOD (28, 3), NUMBER_OF_SENATORS = 100)
```
8.6.12 POINTER statement

R853  pointer-stmt  is  POINTER  [ :: ]  pointer-decl-list
R854  pointer-decl  is  object-name  [ ( deferred-shape-spec-list ) ]
or  proc-entity-name

C887  A  proc-entity-name  shall  have  the  EXTERNAL  attribute.

1  The  POINTER  statement  specifies  the  POINTER  attribute  (8.5.14)  for  a  list  of  entities.

NOTE 8.34
An  example  of  a  POINTER  statement  is:

```fortran
TYPE (NODE) :: CURRENT
POINTER :: CURRENT, A (:, :)
```

8.6.13 PROTECTED statement

R855  protected-stmt  is  PROTECTED  [ :: ]  entity-name-list

1  The  PROTECTED  statement  specifies  the  PROTECTED  attribute  (8.5.15)  for  a  list  of  entities.

8.6.14 SAVE statement

R856  save-stmt  is  SAVE  [ [ :: ]  saved-entity-list ]
R857  saved-entity  is  object-name
or  proc-pointer-name
or  / common-block-name /
R858  proc-pointer-name  is  name

C888  (R856)  If  a  SAVE  statement  with  an  omitted  saved  entity  list  appears  in  a  scoping  unit,  no  other
appearance  of  the  SAVE  attr-spec  or  SAVE  statement  is  permitted  in  that  scoping  unit.

C889  A  proc-pointer-name  shall  be  the  name  of  a  procedure  pointer.

1  A  SAVE  statement  with  a  saved  entity  list  specifies  the  SAVE  attribute  (8.5.16)  for  a  list  of  entities.  A  SAVE
statement  without  a  saved  entity  list  is  treated  as  though  it  contained  the  names  of  all  allowed  items  in  the  same
scoping  unit.

NOTE 8.35
An  example  of  a  SAVE  statement  is:

```fortran
SAVE A, B, C, / BLOCKA /, D
```

8.6.15 TARGET statement

R859  target-stmt  is  TARGET  [ :: ]  target-decl-list
R860  target-decl  is  object-name  [ ( array-spec ) ]
   [ lbracket coarray-spec rbracket ]

1  The  TARGET  statement  specifies  the  TARGET  attribute  (8.5.17)  for  a  list  of  objects.
NOTE 8.36
An example of a TARGET statement is:

```
TARGET :: A (1000, 1000), B
```

### 8.6.16 VALUE statement

R861 `value-stmt` is `VALUE [ :: ] dummy-arg-name-list`

1. The VALUE statement specifies the VALUE attribute (8.5.18) for a list of dummy arguments.

### 8.6.17 VOLATILE statement

R862 `volatile-stmt` is `VOLATILE [ :: ] object-name-list`

1. The VOLATILE statement specifies the VOLATILE attribute (8.5.19) for a list of objects.

### 8.7 IMPLICIT statement

1. In a scoping unit, an IMPLICIT statement specifies a type, and possibly type parameters, for all implicitly typed data entities whose names begin with one of the letters specified in the statement. An IMPLICIT NONE statement can indicate that no implicit typing rules are to apply in a particular scoping unit, or that external and dummy procedures need to be explicitly given the EXTERNAL attribute.

R863 `implicit-stmt` is `IMPLICIT implicit-spec-list`

or `IMPLICIT NONE [ ( [ implicit-none-spec-list ] ) ]`

R864 `implicit-spec` is `declaration-type-spec ( letter-spec-list )`

R865 `letter-spec` is `letter [ – letter ]`

R866 `implicit-none-spec` is `EXTERNAL`

or `TYPE`

C890 (R863) If an IMPLICIT NONE statement appears in a scoping unit, it shall precede any PARAMETER statements that appear in the scoping unit. No more than one IMPLICIT NONE statement shall appear in a scoping unit.

C891 If an IMPLICIT NONE statement in a scoping unit has an implicit-none-spec of TYPE or has no implicit-none-spec-list, there shall be no other IMPLICIT statements in the scoping unit.

C892 (R865) If the minus and second letter appear, the second letter shall follow the first letter alphabetically.

C893 If IMPLICIT NONE with an implicit-none-spec of EXTERNAL appears within a scoping unit, the name of an external or dummy procedure in that scoping unit or in a contained subprogram or BLOCK construct shall be explicitly declared to have the EXTERNAL attribute.

2 A letter-spec consisting of two letters separated by a minus is equivalent to writing a list containing all of the letters in alphabetical order in the alphabetic sequence from the first letter through the second letter. For example, A–C is equivalent to A, B, C. The same letter shall not appear as a single letter, or be included in a range of letters, more than once in all of the IMPLICIT statements in a scoping unit.

3 In each scoping unit, there is a mapping, which may be null, between each of the letters A, B, ..., Z and a type (and type parameters). An IMPLICIT statement specifies the mapping for the letters in its letter-spec-list. IMPLICIT NONE with an implicit-none-spec of TYPE or with no implicit-none-spec-list specifies the null mapping for all the letters. If a mapping is not specified for a letter, the default for a program unit or an interface body is default integer if the letter is I, J, ..., or N and default real otherwise, and the default for a BLOCK
construct, internal subprogram, or module subprogram is the mapping in the host scoping unit.

Any data entity that is not explicitly declared by a type declaration statement, is not an intrinsic function, is not a component, and is not accessed by use or host association is declared implicitly to be of the type (and type parameters) mapped from the first letter of its name, provided the mapping is not null. The mapping for the first letter of the data entity shall either have been established by a prior IMPLICIT statement or be the default mapping for the letter. The data entity is treated as if it were declared in an explicit type declaration; if the outermost inclusive scope in which it appears is not a type definition, it is declared in that scope, otherwise it is declared in the host of that scope. An explicit type specification in a FUNCTION statement overrides an IMPLICIT statement for the result of that function.

NOTE 8.37

The following are examples of the use of IMPLICIT statements:

```
MODULE EXAMPLE_MODULE
  IMPLICIT NONE
  ...
  INTERFACE
    FUNCTION FUN (I) ! Not all data entities need to
    INTEGER FUN ! be declared explicitly
  END FUNCTION FUN
  END INTERFACE
  CONTAINS
    FUNCTION JFUN (J) ! All data entities need to
    INTEGER JFUN, J ! be declared explicitly.
    ...
  END FUNCTION JFUN
END MODULE EXAMPLE_MODULE
SUBROUTINE SUB
  IMPLICIT COMPLEX (C)
  C = (3.0, 2.0) ! C is implicitly declared COMPLEX
  ...
  CONTAINS
    SUBROUTINE SUB1
      IMPLICIT INTEGER (A, C)
      C = (0.0, 0.0) ! C is host associated and of
      ! type complex
      Z = 1.0 ! Z is implicitly declared REAL
      A = 2 ! A is implicitly declared INTEGER
      CC = 1 ! CC is implicitly declared INTEGER
      ...
    END SUBROUTINE SUB1
    SUBROUTINE SUB2
      Z = 2.0 ! Z is implicitly declared REAL and
      ! is different from the variable of
      ! the same name in SUB1
      ...
    END SUBROUTINE SUB2
    SUBROUTINE SUB3
      USE EXAMPLE_MODULE ! Accesses integer function FUN
      ! by use association
      Q = FUN (K) ! Q is implicitly declared REAL and
      ...
      ! K is implicitly declared INTEGER
    END SUBROUTINE SUB3
END SUBROUTINE SUB
```
NOTE 8.38
The following is an example of a mapping to a derived type that is inaccessible in the local scope:

```fortran
PROGRAM MAIN
  IMPLICIT TYPE(BLOB) (A)
  TYPE BLOB
    INTEGER :: I
  END TYPE BLOB
  TYPE(BLOB) :: B
  CALL STEVE
  CONTAINS
    SUBROUTINE STEVE
      INTEGER :: BLOB
      ..
      AA = B
      ..
    END SUBROUTINE STEVE
  END PROGRAM MAIN
```

In the subroutine STEVE, it is not possible to explicitly declare a variable to be of type BLOB because BLOB has been given a different meaning, but implicit mapping for the letter A still maps to type BLOB, so AA is of type BLOB.

NOTE 8.39
Implicit typing is not affected by BLOCK constructs. For example, in

```fortran
SUBROUTINE S(N)
  ...
  IF (N>0) THEN
    BLOCK
      NSQP = CEILING (SQRT (DBLE (N)))
    END BLOCK
  END IF
  ...
  IF (N>0) THEN
    BLOCK
      PRINT *,NSQP
    END BLOCK
  END IF
END SUBROUTINE
```
even if the only two appearances of NSQP are within the BLOCK constructs, the scope of NSQP is the whole subroutine S.

NOTE 8.40
In the subprogram

```fortran
SUBROUTINE EXAMPLE (X, Y)
  IMPLICIT NONE (EXTERNAL)
  REAL, EXTERNAL :: G
  REAL :: X, Y
  X = F (Y) ! Invalid: F lacks the EXTERNAL attribute.
  X = G (Y) ! Valid: G has the EXTERNAL attribute.
END SUBROUTINE
```
NOTE 8.40 (cont.)

the referenced function F needs to have the EXTERNAL attribute (8.5.9).

8.8 IMPORT statement

R867 import-stmt is IMPORT [[ :: ] import-name-list ]
  or IMPORT, ONLY : import-name-list
  or IMPORT, NONE
  or IMPORT, ALL

C894 (R867) An IMPORT statement shall not appear in the specification-part of a main-program, external-subprogram, module, or block-data.

C895 (R867) Each import-name shall be the name of an entity in the host scoping unit.

C896 If any IMPORT statement in a scoping unit has an ONLY specifier, all IMPORT statements in that scoping unit shall have an ONLY specifier.

C897 IMPORT, NONE shall not appear in a submodule.

C898 If an IMPORT, NONE or IMPORT, ALL statement appears in a scoping unit, no other IMPORT statement shall appear in that scoping unit.

C899 Within an interface body, an entity that is accessed by host association shall be accessible by host or use association within the host scoping unit, or explicitly declared prior to the interface body.

C8100 An entity whose name appears as an import-name or which is made accessible by an IMPORT, ALL statement shall not appear in any context described in 19.5.1.4 that would cause the host entity of that name to be inaccessible.

1 If the ONLY specifier appears on an IMPORT statement in a scoping unit, an entity is only accessible by host association if its name appears as an import-name in that scoping unit.

2 An IMPORT, NONE statement in a scoping unit specifies that no entities in the host scoping unit are accessible by host association in that scoping unit. This is the default for an interface body for an external or dummy procedure.

3 An IMPORT, ALL statement in a scoping unit specifies that all entities from the host scoping unit are accessible by host association in that scoping unit.

4 If an IMPORT statement with no import-name-list appears in a scoping unit, every entity in the host scoping unit is accessible unless its name appears in a context described in 19.5.1.4 that causes it to be inaccessible. This is the default for a nested scoping unit other than an interface body for an external or dummy procedure.

5 If an IMPORT statement with an import-name-list appears in a scoping unit, each named entity from the host scoping unit is accessible by host association.

NOTE 8.41

The IMPORT, NONE statement can be used to prevent accidental host association:

```fortran
SUBROUTINE s(x,n)
  IMPLICIT NONE
  IMPORT, NONE
  ...
  DO i=1,n ! Forces I to be locally declared.
```
NOTE 8.42
The IMPORT, ALL statement can be used to confirm the default rules and prevent accidental “shadowing” of host entities:

```
SUBROUTINE outer
    REAL x
    ...
CONTAINS
    SUBROUTINE inner
        IMPORT, ALL
        ...
        x = x + 1 ! X was prevented from being locally declared...
        ! so this is the host X.
```

NOTE 8.43
The IMPORT, ONLY statement can be used to document deliberate access via host association whilst blocking accidental access:

```
SUBROUTINE sub
    IMPORT,ONLY :: x, y
    ...
    x = y + z ! Only X and Y are imported, so Z is local.
```

NOTE 8.44
The IMPORT statement can be used to allow module procedures to have dummy arguments that are procedures with assumed-shape arguments of an opaque type. For example:

```
MODULE M
    TYPE T
        PRIVATE ! T is an opaque type
        ...
    END TYPE
    CONTAINS
    SUBROUTINE PROCESS(X,Y,RESULT,MONITOR)
        TYPE(T),INTENT(IN) :: X(:,,:),Y(:,:)
        TYPE(T),INTENT(OUT) :: RESULT(:,:)
        INTERFACE
            SUBROUTINE MONITOR(ITERATION_NUMBER,CURRENT_ESTIMATE)
                TYPE(T),INTENT(IN) :: ITERATION_NUMBER
                TYPE(T),INTENT(IN) :: CURRENT_ESTIMATE(:,:)
            END SUBROUTINE
        END INTERFACE
    END SUBROUTINE
END MODULE
```

The MONITOR dummy procedure requires an explicit interface because it has an assumed-shape array argument, but TYPE(T) would not be available inside the interface body without the IMPORT statement.
8.9 NAMELIST statement

A NAMELIST statement specifies a group of named data objects, which may be referred to by a single name for the purpose of data transfer (12.6, 13.11).

R868 namelist-stmt is NAMELIST
  / namelist-group-name / namelist-group-object-list 
  [ [ , ] / namelist-group-name / ]
  namelist-group-object-list ] ...

C8101 (R868) The namelist-group-name shall not be a name accessed by use association.

C8102 (R869) A namelist-group-object shall not be an assumed-size array.

C8103 (R868) A namelist-group-object shall not have the PRIVATE attribute if the namelist-group-name has the PUBLIC attribute.

The order in which the values appear on output is the same as the order of the namelist-group-objects in the namelist group object list; if a variable appears more than once as a namelist-group-object for the same namelist group, its value appears once for each occurrence.

Any namelist-group-name may occur more than once in the NAMELIST statements in a scoping unit. The namelist-group-object-list following each successive appearance of the same namelist-group-name in a scoping unit is treated as a continuation of the list for that namelist-group-name.

A namelist group object may be a member of more than one namelist group.

A namelist group object shall either be accessed by use or host association or shall have its declared type, kind type parameters of the declared type, and rank specified by previous specification statements or the procedure heading in the same scoping unit or by the implicit typing rules in effect for the scoping unit. If a namelist group object is typed by the implicit typing rules, its appearance in any subsequent type declaration statement shall confirm the implied type and type parameters.

NOTE 8.45
An example of a NAMELIST statement is:

```
NAMELIST /NLIST/ A, B, C
```

8.10 Storage association of data objects

8.10.1 EQUIVALENCE statement

8.10.1.1 General

An EQUIVALENCE statement is used to specify the sharing of storage units by two or more objects in a scoping unit. This causes storage association (19.5.3) of the objects that share the storage units.

If the equivalenced objects have differing type or type parameters, the EQUIVALENCE statement does not cause type conversion or imply mathematical equivalence. If a scalar and an array are equivalenced, the scalar does not have array properties and the array does not have the properties of a scalar.

R870 equivalence-stmt is EQUIVALENCE equivalence-set-list
R871 equivalence-set is ( equivalence-object , equivalence-object-list )
R872 equivalence-object is variable-name or array-element or substring

ISO/IEC JTC 1/SC 22/WG5/N2118 125
An equivalence-object shall not be a designator with a base object that is a dummy argument, a function result, a pointer, an allocatable variable, a derived-type object that has an allocatable or pointer ultimate component, an object of a nonsequence derived type, an automatic data object, a coarray, a variable with the BIND attribute, a variable in a common block that has the BIND attribute, or a named constant.

An equivalence-object shall not have the TARGET attribute.

Each subscript or substring range expression in an equivalence-object shall be an integer constant expression.

An equivalence-object shall not be a designator that has more than one part-ref.

An equivalence-object shall not have the TARGET attribute.

An equivalence-object shall not be a designator with a base object that is a dummy argument, a function result, a pointer, an allocatable variable, a derived-type object that has an allocatable or pointer ultimate component, an object of a nonsequence derived type, an automatic data object, a coarray, a variable with the BIND attribute, a variable in a common block that has the BIND attribute, or a named constant.

NOTE 8.46

The EQUIVALENCE statement allows the equivalencing of sequence structures and the equivalencing of objects of intrinsic type with nondefault type parameters, but there are strict rules regarding the appearance of these objects in an EQUIVALENCE statement.

In addition to the above constraints, further rules on the interaction of EQUIVALENCE statements and default initialization are given in 19.5.3.4.

8.10.1.2 Equivalence association

An EQUIVALENCE statement specifies that the storage sequences (19.5.3.2) of the data objects specified in an equivalence-set are storage associated. All of the nonzero-sized sequences in the equivalence-set, if any, have the same first storage unit, and all of the zero-sized sequences in the equivalence-set, if any, are storage associated with one another and with the first storage unit of any nonzero-sized sequences. This causes the storage association of the data objects in the equivalence-set and may cause storage association of other data objects.

If any data object in an equivalence-set has the SAVE attribute, all other objects in the equivalence-set have the SAVE attribute; this may be confirmed by explicit specification.

8.10.1.3 Equivalence of default character objects

A default character data object shall not be equivalenced to an object that is not default character and not of a character sequence type. The lengths of equivalenced default character objects need not be the same.

An EQUIVALENCE statement specifies that the storage sequences of all the default character data objects specified in an equivalence-set are storage associated. All of the nonzero-sized sequences in the equivalence-set, if any, have the same first character storage unit, and all of the zero-sized sequences in the equivalence-set, if any, are storage associated with one another and with the first character storage unit of any nonzero-sized sequences. This causes the storage association of the data objects in the equivalence-set and may cause storage association of other data objects.

NOTE 8.47

For example, using the declarations:

CHARACTER (LEN = 4) :: A, B
CHARACTER (LEN = 3) :: C (2)
EQUIVALENCE (A, C (1)), (B, C (2))
NOTE 8.47 (cont.)

the association of A, B, and C can be illustrated graphically as:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>---</td>
<td>A</td>
<td>---</td>
<td></td>
<td>---</td>
<td>B</td>
</tr>
</tbody>
</table>

8.10.1.4 Array names and array element designators

1 For a nonzero-sized array, the use of the array name unqualified by a subscript list as an equivalence-object has the same effect as using an array element designator that identifies the first element of the array.

8.10.1.5 Restrictions on EQUIVALENCE statements

1 An EQUIVALENCE statement shall not specify that the same storage unit is to occur more than once in a storage sequence.

2 An EQUIVALENCE statement shall not specify that consecutive storage units are to be nonconsecutive.

8.10.2 COMMON statement

8.10.2.1 General

1 The COMMON statement specifies blocks of physical storage, called common blocks, that can be accessed by any of the scoping units in a program. Thus, the COMMON statement provides a global data facility based on storage association (19.5.3).

2 A common block that does not have a name is called blank common.

R873 common-stmt is COMMON[

- common-block-name / common-block-object-list
- common-block-object-list

...]

R874 common-block-object is variable-name [ ( array-spec ) ]

C8115 (R874) An array-spec in a common-block-object shall be an explicit-shape-spec-list.

C8116 (R874) Only one appearance of a given variable-name is permitted in all common-block-object-lists within a scoping unit.

C8117 (R874) A common-block-object shall not be a dummy argument, a function result, an allocatable variable, a derived-type object with an ultimate component that is allocatable, a procedure pointer, an automatic data object, a variable with the BIND attribute, an unlimited polymorphic pointer, or a coarray.

C8118 (R874) If a common-block-object is of a derived type, the type shall have the BIND attribute or the SEQUENCE attribute and it shall have no default initialization.

C8119 (R874) A variable-name shall not be a name made accessible by use association.

3 In each COMMON statement, the data objects whose names appear in a common block object list following a common block name are declared to be in that common block. If the first common block name is omitted, all data objects whose names appear in the first common block object list are specified to be in blank common. Alternatively, the appearance of two slashes with no common block name for blank common may occur more than once in one or more COMMON statements in a scoping unit. The common block list following each successive appearance of the same common block name in a scoping unit is treated as a continuation of the list for that common block name. Similarly, each blank common block object list in a scoping unit is treated as a continuation of blank common.

4 Any common block name or an omitted common block name for blank common may occur more than once in one or more COMMON statements in a scoping unit. The common block list following each successive appearance of the same common block name in a scoping unit is treated as a continuation of the list for that common block name. Similarly, each blank common block object list in a scoping unit is treated as a continuation of blank common.

5 The form variable-name (array-spec) specifies the DIMENSION attribute for that variable.

6 If derived-type objects of numeric sequence type or character sequence type (7.5.2.3) appear in common, it is as if the individual components were enumerated directly in the common list.

8.10.2.2 Common block storage sequence

1 For each common block in a scoping unit, a common block storage sequence is formed as follows:
A storage sequence is formed consisting of the sequence of storage units in the storage sequences (19.5.3.2) of all data objects in the common block object lists for the common block. The order of the storage sequences is the same as the order of the appearance of the common block object lists in the scoping unit.

The storage sequence formed in (1) is extended to include all storage units of any storage sequence associated with it by equivalence association. The sequence shall be extended only by adding storage units beyond the last storage unit. Data objects associated with an entity in a common block are considered to be in that common block.

Only COMMON statements and EQUIVALENCE statements appearing in the scoping unit contribute to common block storage sequences formed in that scoping unit.

8.10.2.3 Size of a common block

The size of a common block is the size of its common block storage sequence, including any extensions of the sequence resulting from equivalence association.

8.10.2.4 Common association

Within a program, the common block storage sequences of all nonzero-sized common blocks with the same name have the same first storage unit, and the common block storage sequences of all zero-sized common blocks with the same name are storage associated with one another. Within a program, the common block storage sequences of all nonzero-sized blank common blocks have the same first storage unit and the storage sequences of all zero-sized blank common blocks are associated with one another and with the first storage unit of any nonzero-sized blank common blocks. This results in the association of objects in different scoping units. Use or host association may cause these associated objects to be accessible in the same scoping unit.

A nonpointer object that is default integer, default real, double precision real, default complex, default logical, or of numeric sequence type shall be associated only with nonpointer objects of these types and kinds.

A nonpointer object that is default character or of character sequence type shall be associated only with nonpointer objects of these types and kinds.

A nonpointer object of a derived type that is not a numeric sequence or character sequence type shall be associated only with nonpointer objects of the same type with the same type parameter values.

A nonpointer object of intrinsic type but which is not default integer, default real, double precision real, default complex, default logical, or default character shall be associated only with nonpointer objects of the same type and type parameters.

A data pointer shall be storage associated only with data pointers of the same type and rank. Data pointers that are storage associated shall have deferred the same type parameters; corresponding nondeferred type parameters shall have the same value.

An object with the TARGET attribute shall be storage associated only with another object that has the TARGET attribute and the same type and type parameters.

NOTE 8.48
A common block is permitted to contain sequences of different storage units, provided each scoping unit that accesses the common block specifies an identical sequence of storage units for the common block. For example, this allows a single common block to contain both numeric and character storage units.

Association in different scoping units between objects of default type, objects of double precision real type, and sequence structures is permitted according to the rules for equivalence objects (8.10.1).

8.10.2.5 Differences between named common and blank common

A blank common block has the same properties as a named common block, except for the following:

- Execution of a RETURN or END statement might cause data objects in a named common block to become undefined unless the common block has the SAVE attribute, but never causes data objects in blank common to become undefined (19.6.6).
- Named common blocks of the same name shall be of the same size in all scoping units of a program in which they appear, but blank common blocks may be of different sizes.
- A data object in a named common block may be initially defined by means of a DATA statement or type declaration statement in a block data program unit (14.3), but objects in blank common shall not be initially defined.

8.10.3 Restrictions on common and equivalence

An EQUIVALENCE statement shall not cause the storage sequences of two different common blocks to be associated.

Equivalence association shall not cause a derived-type object with default initialization to be associated with an object in a common block.
Equivalence association shall not cause a common block storage sequence to be extended by adding storage units preceding the first storage unit of the first object specified in a COMMON statement for the common block.
9 Use of data objects

9.1 Designator

R901 designator is object-name or array-element or array-section or coindexed-named-object or complex-part-designator or structure-component or substring

The appearance of a data object designator in a context that requires its value is termed a reference.

9.2 Variable

R902 variable is designator or function-reference

C901 (R902) designator shall not be a constant or a subobject of a constant.

C902 (R902) function-reference shall have a data pointer result.

A variable is either the data object denoted by designator or the target of the pointer resulting from the evaluation of function-reference; this pointer shall be associated.

A reference is permitted only if the variable is defined. A reference to a data pointer is permitted only if the pointer is associated with a target object that is defined. A variable becomes defined with a value when events described in 19.6.5 occur.

R903 variable-name is name

C903 (R903) variable-name shall be the name of a variable.

R904 logical-variable is variable

C904 (R904) logical-variable shall be of type logical.

R905 char-variable is variable

C905 (R905) char-variable shall be of type character.

R906 default-char-variable is variable

C906 (R906) default-char-variable shall be default character.

R907 int-variable is variable

C907 (R907) int-variable shall be of type integer.

NOTE 9.1

For example, given the declarations:

CHARACTER (10) A, B (10)
NOTE 9.1 (cont.)

\begin{verbatim}
TYPE (PERSON) P ! See Note 7.15

then A, B, B (1), B (1:5), P % AGE, and A (1:1) are all variables.
\end{verbatim}

9.3 Constants

1 A constant (6.2.3) is a literal constant or a named constant. A literal constant is a scalar denoted by a syntactic form, which indicates its type, type parameters, and value. A named constant is a constant that has a name; the name has the PARAMETER attribute (8.5.13, 8.6.11). A reference to a constant is always permitted; redefinition of a constant is never permitted.

9.4 Scalars

9.4.1 Substrings

1 A substring is a contiguous portion of a character string (7.4.4).

\begin{verbatim}
R908  substring is parent-string ( substring-range )
R909  parent-string is scalar-variable-name
      or array-element
      or coindexed-named-object
      or scalar-structure-component
      or scalar-constant
R910  substring-range is [ scalar-int-expr ] : [ scalar-int-expr ]
C908  (R909) parent-string shall be of type character.
\end{verbatim}

2 The value of the first scalar-int-expr in substring-range is the starting point of the substring and the value of the second one is the ending point of the substring. The length of a substring is the number of characters in the substring and is \( \max (l - f + 1, 0) \), where \( f \) and \( l \) are the starting and ending points, respectively.

3 Let the characters in the parent string be numbered 1, 2, 3, ..., \( n \), where \( n \) is the length of the parent string. Then the characters in the substring are those from the parent string from the starting point and proceeding in sequence up to and including the ending point. If the starting point is greater than the ending point, the substring has length zero; otherwise, both the starting point and the ending point shall be within the range 1, 2, ..., \( n \). If the starting point is not specified, the default value is 1. If the ending point is not specified, the default value is \( n \).

NOTE 9.2

Examples of character substrings are:

\begin{verbatim}
B(1)(1:5)    array element as parent string
P%NAME(1:1)  structure component as parent string
ID(4:9)      scalar variable name as parent string
'0123456789'(N:N) character constant as parent string
\end{verbatim}

9.4.2 Structure components

1 A structure component is part of an object of derived type; it may be referenced by an object designator. A structure component may be a scalar or an array.

\begin{verbatim}
R911  data-ref is part-ref [ % part-ref ] ...
\end{verbatim}
R912  `part-ref` is `part-name` [(section-subscript-list)] [image-selector]

C909 (R911) Each `part-name` except the rightmost shall be of derived type.

C910 (R911) Each `part-name` except the leftmost shall be the name of a component of the declared type of the preceding `part-name`.

C911 (R911) If the rightmost `part-name` is of abstract type, `data-ref` shall be polymorphic.

C912 (R911) The leftmost `part-name` shall be the name of a data object.

C913 (R912) If a section-subscript-list appears, the number of section-subscripts shall equal the rank of `part-name`.

C914 (R912) If image-selector appears, the number of cosubscripts shall be equal to the corank of `part-name`.

C915 (R912) If image-selector appears and `part-name` is an array, section-subscript-list shall appear.

C916 (R911) Except as an actual argument to an intrinsic inquiry function or as the designator in a type parameter inquiry, a `data-ref` shall not be a polymorphic subobject of a coindexed object and shall not be a coindexed object that has a polymorphic allocatable potential subobject component.

The **rank** of a `part-ref` of the form `part-name` is the **rank** of `part-name`. The **rank** of a `part-ref` that has a section subscript list is the number of subscript triplets and vector subscripts in the list.

C917 (R911) There shall not be more than one `part-ref` with nonzero **rank**. A `part-name` to the right of a `part-ref` with nonzero **rank** shall not have the ALLOCATABLE or POINTER attribute.

The **rank** of a `data-ref` is the **rank** of the `part-ref` with nonzero **rank**, if any; otherwise, the **rank** is zero. The base object of a `data-ref` is the data object whose name is the leftmost part name.

The type and type parameters, if any, of a `data-ref` are those of the rightmost part name.

A `data-ref` with more than one `part-ref` is a subobject of its base object if none of the part-names, except for possibly the rightmost, is a pointer. If the rightmost part-name is the only pointer, then the `data-ref` is a subobject of its base object in contexts that pertain to its pointer association status but not in any other contexts.

**NOTE 9.3**

If X is an object of derived type with a pointer component P, then the pointer X%P is a subobject of X when considered as a pointer – that is in contexts where it is not dereferenced.

However the target of X%P is not a subobject of X. Thus, in contexts where X%P is dereferenced to refer to the target, it is not a subobject of X.

R913  `structure-component` is `data-ref`

C918 (R913) There shall be more than one `part-ref` and the rightmost `part-ref` shall not have a section-subscript-list.

A `structure component` shall be neither referenced nor defined before the declaration of the base object. A `structure component` is a pointer only if the rightmost part name has the POINTER attribute.

**NOTE 9.4**

Examples of structure components are:

- `SCALAR_PARENT%SCALAR_FIELD` scalar component of scalar parent
- `ARRAY_PARENT(J)%SCALAR_FIELD` component of array element parent
- `ARRAY_PARENT(1:N)%SCALAR_FIELD` component of array section parent
NOTE 9.4 (cont.)
For a more elaborate example see C.4.1.

NOTE 9.5
The syntax rules are structured such that a *data-ref* that ends in a component name without a following subscript list is a structure component, even when other component names in the *data-ref* are followed by a subscript list. A *data-ref* that ends in a component name with a following subscript list is either an array element or an array section. A *data-ref* of nonzero rank that ends with a *substring-range* is an array section. A *data-ref* of zero rank that ends with a *substring-range* is a substring.

9.4.3 Coindexed named objects
1 A coindexed-named-object is a named scalar coarray variable followed by an image selector.

R914 coindexed-named-object is data-ref

C919 (R914) The *data-ref* shall contain exactly one part-ref. The part-ref shall contain an image-selector. The part-name shall be the name of a scalar coarray.

9.4.4 Complex parts
7 R915 complex-part-designator is designator % RE or designator % IM

C920 (R915) The designator shall be of complex type.

1 If complex-part-designator is designator%RE it designates the real part of designator. If it is designator%IM it designates the imaginary part of designator. The type of a complex-part-designator is real, and its kind and shape are those of the designator, which can be an array or scalar.

NOTE 9.6
The following are examples of complex part designators:

```fortran
impedance%re !-- Same value as REAL(impedance)
fft%im !-- Same value as AIMAG(fft)
x%im = 0.0 !-- Sets the imaginary part of X to zero
```

9.4.5 Type parameter inquiry
1 A type parameter inquiry is used to inquire about a type parameter of a data object. It applies to both intrinsic and derived types.

R916 type-param-inquiry is designator % type-param-name

C921 (R916) The type-param-name shall be the name of a type parameter of the declared type of the object designated by the designator.

2 A deferred type parameter of a pointer that is not associated or of an unallocated allocatable variable shall not be inquired about.

NOTE 9.7
A type-param-inquiry has a syntax like that of a structure component reference, but it does not have the same semantics. It is not a variable and thus can never be assigned to. It can be used only as a primary in an expression. It is scalar even if designator is an array.
NOTE 9.7 (cont.)

The intrinsic type parameters can also be inquired about by using the intrinsic functions KIND and LEN.

NOTE 9.8

The following are examples of type parameter inquiries:

\[
\begin{align*}
    a \% \text{kind} & \quad !-- A \text{ is real. Same value as KIND(a)}. \\
    s \% \text{len} & \quad !-- S \text{ is character. Same value as LEN(s)}. \\
    b(10) \% \text{kind} & \quad !-- \text{Inquiry about an array element}. \\
    p \% \text{dim} & \quad !-- P \text{ is of the derived type general_point}. \\
\end{align*}
\]

See Note 7.22 for the definition of the \texttt{general_point} type used in the last example above.

9.5 Arrays

9.5.1 Order of reference

1. No order of reference to the elements of an array is indicated by the appearance of the array designator, except where array element ordering (9.5.3.2) is specified.

9.5.2 Whole arrays

1. A whole array is a named array or a structure component whose final part-ref is an array component name; no subscript list is appended.

2. The appearance of a whole array variable in an executable construct specifies all the elements of the array (5.4.6). The appearance of a whole array designator in a nonexecutable statement specifies the entire array except for the appearance of a whole array designator in an equivalence set (8.10.1.4). An assumed-size array (8.5.8.5) is permitted to appear as a whole array in an executable construct or specification expression only as an actual argument in a procedure reference that does not require the shape.

9.5.3 Array elements and array sections

9.5.3.1 Syntax

R917 array-element is data-ref

C922 (R917) Every part-ref shall have rank zero and the last part-ref shall contain a subscript-list.

R918 array-section is data-ref [ { substring-range } ]

or complex-part-designator

C923 (R918) Exactly one part-ref shall have nonzero rank, and either the final part-ref shall have a section-subscript-list with nonzero rank, another part-ref shall have nonzero rank, or the complex-part-designator shall be an array.

C924 (R918) If a substring-range appears, the rightmost part-name shall be of type character.

R919 subscript is scalar-int-expr

R920 section-subscript is subscript

or subscript-triplet

or vector-subscript

R921 subscript-triplet is [ subscript ] : [ subscript ] [ : stride ]
1. \textit{stride} \textit{is} \textit{scalar-int-expr}

2. \textit{vector-subscript} \textit{is} \textit{int-expr}

3. \textit{C925} (R923) A \textit{vector-subscript} \textit{shall} be an integer array expression of \textit{rank one}.

4. \textit{C926} (R921) The second subscript \textit{shall} not be omitted from a \textit{subscript-triplet} \textit{in} the last dimension of an \textit{assumed-size array}.

1. An array element \textit{is} a scalar. An \textit{array section} \textit{is} an array. If a \textit{substring-range} \textit{appears} in an \textit{array-section}, each element \textit{is} the designated substring of the corresponding element \textit{of} the \textit{array section}.

2. The value of a subscript \textit{in} an array element \textit{shall} be \textit{within} the bounds \textit{for} its dimension.

\textbf{NOTE 9.9}

For example, \textit{with} the declarations:

\begin{verbatim}
REAL A (10, 10)
CHARACTER (LEN = 10) B (5, 5, 5)
\end{verbatim}

A (1, 2) \textit{is an array element}, A (1:N:2, M) \textit{is a rank-one array section}, and B (:, ;, :) (2:3) \textit{is an array} \textit{of shape} (5, 5, 5) \textit{whose elements are substrings of length} 2 \textit{of} the corresponding elements \textit{of} B.


\textbf{NOTE 9.10}

Unless \textit{otherwise} specified, an array element or \textit{array section} \textit{does not} have an attribute \textit{of} the \textit{whole array}. \textit{In particular}, an array element or an \textit{array section} \textit{does not} have the \textit{POINTER} or \textit{ALLOCATABLE} attribute.

\textbf{NOTE 9.11}

Examples \textit{of} array elements \textit{and} array sections \textit{are:}

\begin{verbatim}
ARRAY_A(1:N:2)%ARRAY_B(I, J)%STRING(K)(): array section
SCALAR_PARENT%ARRAY_FIELD(J) array element
SCALAR_PARENT%ARRAY_FIELD(1:N) array section
SCALAR_PARENT%ARRAY_FIELD(1:N)%SCALAR_FIELD array section
\end{verbatim}

9.5.3.2 \textit{Array element order}

1. The elements \textit{of an array} \textit{form} a sequence \textit{known as the array element order}. \textit{The} position \textit{of an array element} \textit{in} this sequence \textit{is determined by} \textit{the subscript order value} \textit{of the subscript list designating the element}. The subscript order value \textit{is computed from} the formulas \textit{in Table 9.1}.

\begin{table}[h]
\centering
\caption{\textit{Subscript order value}}
\begin{tabular}{|c|c|c|}
\hline
\textbf{Rank} & \textbf{Subscript bounds} & \textbf{Subscript list} & \textbf{Subscript order value} \\
\hline
1 & $j_1:k_1$ & $s_1$ & $1 + (s_1 - j_1)$ \\
\hline
2 & $j_1:k_1,j_2:k_2$ & $s_1,s_2$ & $1 + (s_1 - j_1)$ \\
& & & $+ (s_2 - j_2) \times d_1$ \\
\hline
3 & $j_1:k_1,j_2:k_2,j_3:k_3$ & $s_1,s_2,s_3$ & $1 + (s_1 - j_1)$ \\
& & & $+ (s_2 - j_2) \times d_1$ \\
& & & $+ (s_3 - j_3) \times d_2 \times d_1$ \\
\hline
\end{tabular}
\end{table}

\textit{ISO/IEC JTC 1/SC 22/WG5/N2118}
Subscript order value (cont.)

<table>
<thead>
<tr>
<th>Rank</th>
<th>Subscript bounds</th>
<th>Subscript list</th>
<th>Subscript order value</th>
</tr>
</thead>
</table>
| 15   | $j_1 : k_1, \ldots, j_{15} : k_{15}$ | $s_1, \ldots, s_{15}$ | $1 + (s_1 - j_1)$  
+ $(s_2 - j_2) \times d_1$  
+ $(s_3 - j_3) \times d_2 \times d_1$  
+ $\ldots$  
+ $(s_{15} - j_{15}) \times d_{14}$  
$\times d_{13} \times \ldots \times d_1$ |

Notes for Table 9.1:

1) $d_i = \max (k_i - j_i + 1, 0)$ is the size of the $i$th dimension.
2) If the size of the array is nonzero, $j_i \leq s_i \leq k_i$ for all $i = 1, 2, \ldots, 15$.

9.5.3.3 Array sections

1 In an array-section having a section-subscript-list, each subscript-triplet and vector-subscript in the section subscript list indicates a sequence of subscripts, which may be empty. Each subscript in such a sequence shall be within the bounds for its dimension unless the sequence is empty. The array section is the set of elements from the array determined by all possible subscript lists obtainable from the single subscripts or sequences of subscripts specified by each section subscript.

2 In an array-section with no section-subscript-list, the rank and shape of the array is the rank and shape of the part-ref with nonzero rank; otherwise, the rank of the array section is the number of subscript triplets and vector subscripts in the section subscript list. The shape is the rank-one array whose $i$th element is the number of integer values in the sequence indicated by the $i$th subscript triplet or vector subscript. If any of these sequences is empty, the array section has size zero. The subscript order of the elements of an array section is that of the array data object that the array section represents.

9.5.3.3.1 Subscript triplet

1 A subscript triplet designates a regular sequence of subscripts consisting of zero or more subscript values. The stride in the subscript triplet specifies the increment between the subscript values. The subscripts and stride of a subscript triplet are optional. An omitted first subscript in a subscript triplet is equivalent to a subscript whose value is the lower bound for the array and an omitted second subscript is equivalent to the upper bound. An omitted stride is equivalent to a stride of 1.

2 The stride shall not be zero.

3 When the stride is positive, the subscripts specified by a triplet form a regularly spaced sequence of integers beginning with the first subscript and proceeding in increments of the stride to the largest such integer not greater than the second subscript; the sequence is empty if the first subscript is greater than the second.

NOTE 9.12

For example, suppose an array is declared as $A(5, 4, 3)$. The section $A(3 : 5, 2, 1 : 2)$ is the array of shape $(3, 2)$:

\[
\begin{array}{c}
A(3, 2, 1) \\
A(4, 2, 1) \\
A(5, 2, 1)
\end{array}
\quad
\begin{array}{c}
A(3, 2, 2) \\
A(4, 2, 2) \\
A(5, 2, 2)
\end{array}
\]

4 When the stride is negative, the sequence begins with the first subscript and proceeds in increments of the stride down to the smallest such integer equal to or greater than the second subscript; the sequence is empty if the second subscript is greater than the first.
NOTE 9.13

For example, if an array is declared B (10), the section B (9 : 1 : 2) is the array of shape (5) whose elements are B (9), B (7), B (5), B (3), and B (1), in that order.

NOTE 9.14

A subscript in a subscript triplet need not be within the declared bounds for that dimension if all values used in selecting the array elements are within the declared bounds.

For example, if an array is declared as B (10), the array section B (3 : 11 : 7) is the array of shape (2) consisting of the elements B (3) and B (10), in that order.

9.5.3.3.2 Vector subscript

A vector subscript designates a sequence of subscripts corresponding to the values of the elements of the expression. Each element of the expression shall be defined.

An array section with a vector subscript shall not be finalized by a nonelemental final subroutine.

If a vector subscript has two or more elements with the same value, an array section with that vector subscript is not definable and shall not be defined or become undefined.

NOTE 9.15

For example, suppose Z is a two-dimensional array of shape [5, 7] and U and V are one-dimensional arrays of shape (3) and (4), respectively. Assume the values of U and V are:

\[ U = [1, 3, 2] \]
\[ V = [2, 1, 1, 3] \]

Then Z (3, V) consists of elements from the third row of Z in the order:

Z (3, 2) Z (3, 1) Z (3, 1) Z (3, 3)

and Z (U, 2) consists of the column elements:

Z (1, 2) Z (3, 2) Z (2, 2)

and Z (U, V) consists of the elements:

Z (1, 2) Z (1, 1) Z (1, 1) Z (1, 3)
Z (3, 2) Z (3, 1) Z (3, 1) Z (3, 3)
Z (2, 2) Z (2, 1) Z (2, 1) Z (2, 3)

Because Z (3, V) and Z (U, V) contain duplicate elements from Z, the sections Z (3, V) and Z (U, V) shall not be redefined as sections.

9.5.4 Simply contiguous array designators

A section-subscript-list specifies a simply contiguous section if and only if it does not have a vector subscript and

- all but the last subscript-triplet is a colon,
- the last subscript-triplet does not have a stride, and
- no subscript-triplet is preceded by a section-subscript that is a subscript.

An array designator is simply contiguous if and only if it is

- an object-name that has the CONTIGUOUS attribute,
An array variable is simply contiguous if and only if it is a simply contiguous array designator or a reference to a function that returns a pointer with the CONTIGUOUS attribute.

NOTE 9.16

Array sections that are simply contiguous include column, plane, cube, and hypercube subobjects of a simply contiguous base object, for example:

```
ARRAY1 (10:20, 3) ! passes part of the third column of ARRAY1.
X3D (:, i:j, 2) ! passes part of the second plane of X3D (or the whole plane if i==LBOUND(X3D,2) and j==UBOUND(X3D,2).
Y5D (:, :, :, :, 7) ! passes the seventh hypercube of Y5D.
```

All simply contiguous designators designate contiguous objects.

9.6 Image selectors

1 An image selector determines the image index for a coindexed object.

R924 image-selector is [ , image-selector-spec-list ] rbracket

R925 cosubscript is scalar-int-expr

R926 image-selector-spec is STAT = stat-variable

or TEAM = team-variable

or TEAM_NUMBER = scalar-int-expr

C927 No specifier shall appear more than once in a given image-selector-spec-list.

C928 TEAM and TEAM_NUMBER shall not both appear in the same image-selector-spec-list.

C929 A stat-variable in an image-selector shall not be a coindexed object.

2 The number of cosubscripts shall be equal to the corank of the object. The value of a cosubscript in an image selector shall be within the cobounds for its codimension. Taking account of the cobounds, the cosubscript list in an image selector determines the image index in the same way that a subscript list in an array element determines the subscript order value (9.5.3.2), taking account of the bounds.

C927 If a TEAM= specifier appears in an image-selector, the team of the image selector is specified by team-variable, which shall be defined with a value that represents the current or an ancestor team; the object shall be an established coarray in that team or an ancestor thereof. If a TEAM_NUMBER= specifier appears in image-selector and the current team is not the initial team, the value of the scalar-int-expr shall be equal to the value of a team number for one of the teams that were formed by execution of the FORM TEAM statement for the
current team and the team of the image selector is that team; the object shall be an established coarray in an
ancestor of the current team. Otherwise, the team of the image selector is the current team.

An image selector shall specify an image index value that is not greater than the number of images in the team
of the image selector, and identifies the image with that index in that team.

Execution of a statement containing an image-selector with a STAT= specifier causes the stat-variable to become
defined. If the designator is part of an operand that is evaluated or is a variable that is being defined or partly
defined, and the object designated is on a failed image, the stat-variable is defined with the value STAT_FAILED_IMAGE (16.10.2.26) in the intrinsic module ISO_FORTRAN_ENV; otherwise, it is defined with the value zero.

The denotation of a stat-variable in an image-selector shall not depend on the evaluation of any entity in the
same statement. The value of an expression shall not depend on the value of any stat-variable that appears in
the same statement. The value of a stat-variable in an image-selector shall not be affected by the execution of
any part of the statement, other than by whether the image specified by the image-selector has failed.

NOTE 9.17
For example, if there are 16 images and the coarray A is declared

\[
\text{REAL} :: A(10) \{ 5, * \}
\]
A(\cdot)[1,4] is valid because it specifies image 16, but A(\cdot)[2,4] is invalid because it specifies image 17.

9.7 Dynamic association

9.7.1 ALLOCATE statement

9.7.1.1 Form of the ALLOCATE statement

The ALLOCATE statement dynamically creates pointer targets and allocatable variables.

140 ISO/IEC JTC 1/SC 22/WG5/N2118
allocate-coarray-spec  is  [ allocate-coshape-spec-list , ] [ lower-bound-expr : ] *
allocate-coshape-spec  is  [ lower-bound-expr : ] upper-bound-expr

(R933) Each allocate-object shall be a data pointer or an allocatable variable.

(R927) If any allocate-object has a deferred type parameter, is unlimited polymorphic, or is of abstract type, either type-spec or source-expr shall appear.

(R927) If type-spec appears, it shall specify a type with which each allocate-object is type compatible.

(R927) A type-param-value in a type-spec shall be an asterisk if and only if each allocate-object is a dummy argument for which the corresponding type parameter is assumed.

(R927) If type-spec appears, the kind type parameter values of each allocate-object shall be the same as the corresponding type parameter values of the type-spec.

(R927) type-spec shall not specify a type that has a coarray ultimate component.

(R927) If an allocate-object is a coarray, type-spec shall not specify the type C_PTR or C_FUNPTR from the intrinsic module ISO_C_BINDING.

(R927) If an allocate-object is an array, either allocate-shape-spec-list shall appear in its allocation, or source-expr shall appear in the ALLOCATE statement and have the same rank as the allocate-object.

(R932) If allocate-object is scalar, allocate-shape-spec-list shall not appear.

(R932) An allocate-coarray-spec shall appear if and only if the allocate-object is a coarray.

(R932) The number of allocate-shapes in an allocate-shape-spec-list shall be the same as the rank of the allocate-object. The number of allocate-coshapes in an allocate-coarray-spec shall be one less than the corank of the allocate-object.

(R928) No alloc-opt shall appear more than once in a given alloc-opt-list.

(R927) At most one of source-expr and type-spec shall appear.

(R927) Each allocate-object shall be type compatible (7.3.2.3) with source-expr. If SOURCE= appears, source-expr shall be a scalar or have the same rank as each allocate-object.

(R927) If source-expr appears, the kind type parameters of each allocate-object shall have the same values as the corresponding type parameters of source-expr.

(R927) The declared type of source-expr shall not be C_PTR or C_FUNPTR from the intrinsic module ISO_C_BINDING if an allocate-object is a coarray.

(R927) If SOURCE= appears, the declared type of source-expr shall not be EVENT_TYPE or LOCK_TYPE from the intrinsic module ISO_FORTRAN_ENV, or have a potential subobject component of type EVENT_TYPE or LOCK_TYPE.

(R931) The declared type of source-expr shall not have a coarray ultimate component.

(R933) An allocate-object shall not be a coindexed object.

**NOTE 9.18**

If a coarray is of a derived type that has an allocatable component, the component shall be allocated by its own image:

```
TYPE(SOMETHING), ALLOCATABLE :: T[:]
```
NOTE 9.18 (cont.)

| ALLOCATE(T[*])   | ! Allowed - implies synchronization |
| ALLOCATE(T%AAC(N)) | ! Allowed - allocated by its own image |
| ALLOCATE(T[Q]%AAC(N)) | ! Not allowed, because it is not necessarily executed on image Q. |

2 An allocate-object or a bound or type parameter of an allocate-object shall not depend on the value of stat-variable, the value of errmsg-variable, or on the value, bounds, length type parameters, allocation status, or association status of any allocate-object in the same ALLOCATE statement.

3 source-expr shall not be allocated within the ALLOCATE statement in which it appears; nor shall it depend on the value, bounds, deferred type parameters, allocation status, or association status of any allocate-object in that statement.

4 If an ALLOCATE statement has a SOURCE= specifier and an allocate-object that is a coarray, source-expr shall not have a dynamic type of C_PTR or C_FUNPTR from the intrinsic module ISO_C_BINDING, or EVENT_TYPE or LOCK_TYPE from the intrinsic module ISO_FORTRAN_ENV, or have a subcomponent whose dynamic type is EVENT_TYPE or LOCK_TYPE.

5 If type-spec is specified, each allocate-object is allocated with the specified dynamic type and type parameter values; if source-expr is specified, each allocate-object is allocated with the dynamic type and type parameter values of source-expr; otherwise, each allocate-object is allocated with its dynamic type the same as its declared type.

6 If a type-param-value in a type-spec in an ALLOCATE statement is an asterisk, it denotes the current value of that assumed type parameter. If it is an expression, subsequent redefinition or undefinition of any entity in the expression does not affect the type parameter value.

NOTE 9.19

An example of an ALLOCATE statement is:

```
ALLOCATE (X (N), B (-3 : M, 0:9), STAT = IERR_ALLOC)
```

9.7.1.2 Execution of an ALLOCATE statement

1 When an ALLOCATE statement is executed for an array for which allocate-shape-spec-list is specified, the values of the lower bound and upper bound expressions determine the bounds of the array. Subsequent redefinition or undefinition of any entities in the bound expressions do not affect the array bounds. If the lower bound is omitted, the default value is 1. If the upper bound is less than the lower bound, the extent in that dimension is zero and the array has zero size.

2 When an ALLOCATE statement is executed for a coarray, the values of the lower cobound and upper cobound expressions determine the cobounds of the coarray. Subsequent redefinition or undefinition of any entities in the cobound expressions do not affect the cobounds. If the lower cobound is omitted, the default value is 1. The upper cobound shall not be less than the lower cobound.

3 If an allocation specifies a coarray, its dynamic type and the values of corresponding type parameters shall be the same on every active image in the current team. The values of corresponding bounds and corresponding cobounds shall be the same on those images. If the coarray is a dummy argument, its ultimate argument (15.5.2.3) shall be the same coarray on those images.

4 When an ALLOCATE statement is executed for which an allocate-object is a coarray, there is an implicit synchronization of all active images in the current team. On those images, if no error condition other than STAT_STOPPED_IMAGE or STAT_FAILED_IMAGE occurs, execution of the segment (11.6.2) following the statement is delayed until all other active images in the current team have executed the same statement the same number of times in this team. The coarray shall not become allocated on an image unless it is successfully
allocated on all active images in this team.

NOTE 9.20

When an image executes an ALLOCATE statement, communication is not necessarily involved apart from any required for synchronization. The image allocates its coarray and records how the corresponding coarrays on other images are to be addressed. The processor is not required to detect violations of the rule that the bounds are the same on all images, nor is it responsible for detecting or resolving deadlock problems (such as two images waiting on different ALLOCATE statements).

If source-expr is a pointer, it shall be associated with a target. If source-expr is allocatable, it shall be allocated.

When an ALLOCATE statement is executed for an array with no allocate-shape-spec-list, the bounds of source-expr determine the bounds of the array. Subsequent changes to the bounds of source-expr do not affect the array bounds.

If SOURCE= appears, source-expr shall be conformable with allocation. If an allocate-object is not polymorphic and the source-expr is polymorphic with a dynamic type that differs from its declared type, the value provided for that allocate-object is the ancestor component of the source-expr that has the type of the allocate-object; otherwise the value provided is the value of the source-expr. On successful allocation, if allocate-object and source-expr have the same rank the value of allocate-object becomes the value provided, otherwise the value of each element of allocate-object becomes the value provided. The source-expr is evaluated exactly once for each execution of an ALLOCATE statement.

If MOLD= appears and source-expr is a variable, its value need not be defined.

If type-spec appears and the value of a length type parameter it specifies differs from the value of the corresponding nondeferred type parameter specified in the declaration of any allocate-object, an error condition occurs. If the value of a nondeferred length type parameter of an allocate-object differs from the value of the corresponding type parameter of source-expr, an error condition occurs.

The set of error conditions for an ALLOCATE statement is processor dependent. If an error condition occurs during execution of an ALLOCATE statement that does not contain the STAT= specifier, error termination is initiated. The STAT= specifier is described in 9.7.4. The ERRMSG= specifier is described in 9.7.5.

9.7.1.3 Allocation of allocatable variables

The allocation status of an allocatable entity is one of the following at any time.

- The status of an allocatable variable becomes “allocated” if it is allocated by an ALLOCATE statement, if it is allocated during assignment, or if it is given that status by the intrinsic subroutine MOVE_ALLOC (16.9.137). An allocatable variable with this status may be referenced, defined, or deallocated; allocating it causes an error condition in the ALLOCATE statement. The result of the intrinsic function ALLOCATED (16.9.11) is true for such a variable.

- An allocatable variable has a status of “unallocated” if it is not allocated. The status of an allocatable variable becomes unallocated if it is deallocated (9.7.3) or if it is given that status by the intrinsic subroutine MOVE_ALLOC. An allocatable variable with this status shall not be referenced or defined. It shall not be supplied as an actual argument corresponding to a nonallocatable nonoptional dummy argument, except to certain intrinsic inquiry functions. It may be allocated with the ALLOCATE statement. Deallocating it causes an error condition in the DEALLOCATE statement. The result of the intrinsic function ALLOCATED (16.9.11) is false for such a variable.

At the beginning of execution of a program, allocatable variables are unallocated.

When the allocation status of an allocatable variable changes, the allocation status of any associated allocatable variable changes accordingly. Allocation of an allocatable variable establishes values for the deferred type parameters of all associated allocatable variables.
An unsaved allocatable local variable of a procedure has a status of unallocated at the beginning of each invocation of the procedure. An unsaved allocatable local variable of a construct has a status of unallocated at the beginning of each execution of the construct.

When an object of derived type is created by an ALLOCATE statement, any allocatable ultimate components have an allocation status of unallocated unless the SOURCE= specifier appears and the corresponding component of the source-expr is allocated.

If the evaluation of a function would change the allocation status of a variable 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 allocation status of the variable after evaluation of the expression is processor dependent.

### 9.7.1.4 Allocation of pointer targets

Allocation of a pointer creates an object that implicitly has the TARGET attribute. Following successful execution of an ALLOCATE statement for a pointer, the pointer is associated with the target and may be used to reference or define the target. Additional pointers may become associated with the pointer target or a part of the pointer target by pointer assignment. It is not an error to allocate a pointer that is already associated with a target. In this case, a new pointer target is created as required by the attributes of the pointer and any array bounds, type, and type parameters specified by the ALLOCATE statement. The pointer is then associated with this new target. Any previous association of the pointer with a target is broken. If the previous target had been created by allocation, it becomes inaccessible unless other pointers are associated with it. The intrinsic function ASSOCIATED (16.9.16) may be used to determine whether a pointer that does not have undefined association status is associated.

At the beginning of execution of a function whose result is a pointer, the association status of the result pointer is undefined. Before such a function returns, it shall either associate a target with this pointer or cause the association status of this pointer to become disassociated.

### 9.7.2 NULLIFY statement

R939  

nullify-stmt  
is  NULLIFY ( pointer-object-list )

R940  

pointer-object  
is  variable-name
or  structure-component
or  proc-pointer-name

C949 (R940) Each pointer-object shall have the POINTER attribute.

A pointer-object shall not depend on the value, bounds, or association status of another pointer-object in the same NULLIFY statement.

Execution of a NULLIFY statement causes each pointer-object to become disassociated.

**NOTE 9.21**

When a NULLIFY statement is applied to a polymorphic pointer (7.3.2.3), its dynamic type becomes the same as its declared type.

### 9.7.3 DEALLOCATE statement

#### 9.7.3.1 Form of the DEALLOCATE statement

The DEALLOCATE statement causes allocatable variables to be deallocated; it causes pointer targets to be deallocated and the pointers to be disassociated.

R941  

deallocate-stmt  
is  DEALLOCATE ( allocate-object-list [ , dealloc-opt-list ] )
R942 dealloc-opt is STAT = stat-variable
or ERRMSG = errmsg-variable

C950 (R942) No dealloc-opt shall appear more than once in a given dealloc-opt-list.

An allocate-object shall not depend on the value, bounds, allocation status, or association status of another allocate-object in the same DEALLOCATE statement; it also shall not depend on the value of the stat-variable or errmsg-variable in the same DEALLOCATE statement.

The set of error conditions for a DEALLOCATE statement is processor dependent. If an error condition occurs during execution of a DEALLOCATE statement that does not contain the STAT= specifier, error termination is initiated. The STAT= specifier is described in 9.7.4. The ERRMSG= specifier is described in 9.7.5.

When more than one allocated object is deallocated by execution of a DEALLOCATE statement, the order of deallocation is processor dependent.

NOTE 9.22

An example of a DEALLOCATE statement is:

DEALLOCATE (X, B)

9.7.3.2 Deallocation of allocatable variables

Deallocating an unallocated allocatable variable causes an error condition in the DEALLOCATE statement. Deallocation of an allocatable variable with the TARGET attribute causes the pointer association status of any pointer associated with it to become undefined. An allocatable variable shall not be deallocated if it or any subobject of it is argument associated with a dummy argument or construct associated with an associate name.

When the execution of a procedure is terminated by execution of a RETURN or END statement, an unsaved allocatable local variable of the procedure retains its allocation and definition status if it is a function result or a subobject thereof; otherwise, if it is allocated it will be deallocated.

When a BLOCK construct terminates, any unsaved allocated allocatable local variable of the construct is deallocated.

If an executable construct references a function whose result is allocatable or has an allocatable subobject, and the function reference is executed, an allocatable result and any allocated allocatable subobject of the result is deallocated after execution of the innermost executable construct containing the reference.

If a function whose result is allocatable or has an allocatable subobject is referenced in the specification part of a scoping unit, and the function reference is executed, an allocatable result and any allocated allocatable subobject of the result is deallocated before execution of the executable constructs of the scoping unit.

When a procedure is invoked, any allocated allocatable object that is an actual argument corresponding to an INTENT (OUT) allocatable dummy argument is deallocated; any allocated allocatable object that is a subobject of an actual argument corresponding to an INTENT (OUT) dummy argument is deallocated. If a Fortran procedure that has an INTENT (OUT) allocatable dummy argument is invoked by a C function and the corresponding argument in the C function call is a C descriptor that describes an allocated allocatable variable, the variable is deallocated on entry to the Fortran procedure. If a C function is invoked from a Fortran procedure via an interface with an INTENT (OUT) allocatable dummy argument and the corresponding actual argument in the reference to the C function is an allocated allocatable variable, the variable is deallocated on invocation (before execution of the C function begins).

When an intrinsic assignment statement (10.2.1.3) is executed, any noncoarray allocated allocatable subobject of the variable is deallocated before the assignment takes place.

When a variable of derived type is deallocated, any allocated allocatable subobject is deallocated. If an error condition occurs during deallocation, it is processor dependent whether an allocated allocatable subobject is
If an allocatable component is a subobject of a finalizable object, that object is finalized before the component is automatically deallocated.

When a statement that deallocates a coarray is executed, there is an implicit synchronization of all active images in the current team. On those images, if no error condition other than STAT_STOPPED_IMAGE or STAT_FAILED_IMAGE occurs, execution of the segment (11.6.2) following the statement is delayed until all other active images in the current team have executed the same statement the same number of times in this team.

If an allocate-object is a coarray dummy argument, its ultimate argument (15.5.2.3) shall be the same coarray on those images.

The effect of automatic deallocation is the same as that of a DEALLOCATE statement without a dealloc-opt-list.

NOTE 9.23
In the following example:

```fortran
SUBROUTINE PROCESS
   REAL, ALLOCATABLE :: TEMP(:)
   REAL, ALLOCATABLE, SAVE :: X(:)
   ...
END SUBROUTINE PROCESS
```

on return from subroutine PROCESS, the allocation status of X is preserved because X has the SAVE attribute. TEMP does not have the SAVE attribute, so it will be deallocated if it was allocated. On the next invocation of PROCESS, TEMP will have an allocation status of unallocated.

NOTE 9.24
For example, executing a RETURN, END, or END BLOCK statement, or deallocating an object that has an allocatable subobject, can cause deallocation of a coarray, and thus an implicit synchronization of all images.

9.7.3.3 Deallocation of pointer targets

If a pointer appears in a DEALLOCATE statement, its association status shall be defined. Deallocating a pointer that is disassociated or whose target was not created by an ALLOCATE statement causes an error condition in the DEALLOCATE statement. If a pointer is associated with an allocatable entity, the pointer shall not be deallocated. A pointer shall not be deallocated if its target or any subobject thereof is argument associated with a dummy argument or construct associated with an associate name.

9.7.4 STAT= specifier

The stat-variable shall not be allocated or deallocated within the ALLOCATE or DEALLOCATE statement in which it appears; nor shall it depend on the value, bounds, deferred type parameters, allocation status, or association status of any allocate-object in that statement. The stat-variable shall not depend on the value of the errmsg-variable.

If the STAT= specifier appears, successful execution of the ALLOCATE or DEALLOCATE statement causes the stat-variable to become defined with a value of zero.
If an `ALLOCATE` or `DEALLOCATE` statement with a coarray `allocate-object` is executed when one or more images of the current team has initiated normal termination of execution, the `stat-variable` becomes defined with the processor-dependent positive integer value of the constant `STAT_STOPPED_IMAGE` from the intrinsic module `ISO_FORTRAN_ENV (16.10.2)`. Otherwise, if a failed image is detected and no other error condition occurs, each `allocate-object` is successfully allocated or deallocated on all the active images of the current team and the `stat-variable` becomes defined with the processor-dependent positive integer value of the constant `STAT_FAILED_IMAGE` from the intrinsic module `ISO_FORTRAN_ENV`. If any other error condition occurs during execution of the `ALLOCATE` or `DEALLOCATE` statement, the `stat-variable` becomes defined with a processor-dependent positive integer value different from `STAT_STOPPED_IMAGE` and `STAT_FAILED_IMAGE`. In any case, each `allocate-object` has a processor-dependent status:

- each `allocate-object` that was successfully allocated shall have an allocation status of allocated or a pointer association status of associated;
- each `allocate-object` that was successfully deallocated shall have an allocation status of unallocated or a pointer association status of disassociated;
- each `allocate-object` that was not successfully allocated or deallocated shall retain its previous allocation status or pointer association status.

**NOTE 9.25**
The status of objects that were not successfully allocated or deallocated can be individually checked with the intrinsic functions `ALLOCATED` or `ASSOCIATED`.

### 9.7.5 ERRMSG= specifier

1. The `errmsg-variable` shall not be allocated or deallocated within the `ALLOCATE` or `DEALLOCATE` statement in which it appears; nor shall it depend on the value, bounds, deferred type parameters, allocation status, or association status of any `allocate-object` in that statement. The `errmsg-variable` shall not depend on the value of the `stat-variable`.

2. If an error condition occurs during execution of an `ALLOCATE` or `DEALLOCATE` statement, the `errmsg-variable` is assigned an explanatory message, as if by intrinsic assignment. If no such condition occurs, the definition status and value of `errmsg-variable` are unchanged.
10 Expressions and assignment

10.1 Expressions

10.1.1 Expression semantics

An expression represents either a data object reference or a computation, and its value is either a scalar or an array. Evaluation of an expression produces a value, which has a type, type parameters (if appropriate), and a shape (10.1.9). The corank of an expression that is not a variable is zero.

10.1.2 Form of an expression

10.1.2.1 Overall expression syntax

An expression is formed from operands, operators, and parentheses. An operand is either a scalar or an array. An operation is either intrinsic (10.1.5) or defined (10.1.6). More complicated expressions can be formed using operands which are themselves expressions.

10.1.2.2 Primary

R1001 primary is literal-constant or designator or array-constructor or structure-constructor or function-reference or type-param-inquiry or type-param-name or ( expr )

C1001 (R1001) The type-param-name shall be the name of a type parameter.

C1002 (R1001) The designator shall not be a whole assumed-size array.

C1003 (R1001) The expr shall not be a function reference that returns a procedure pointer.

NOTE 10.1

Examples of a primary are:

<table>
<thead>
<tr>
<th>Example</th>
<th>Syntactic class</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>constant</td>
</tr>
<tr>
<td>'ABCDEFGHJKLMNPQRSTUVWXYZ' (I:1)</td>
<td>designator</td>
</tr>
<tr>
<td>[ 1.0, 2.0 ]</td>
<td>array-constructor</td>
</tr>
<tr>
<td>PERSON ('Jones', 12)</td>
<td>structure-constructor</td>
</tr>
<tr>
<td>F (X, Y)</td>
<td>function-reference</td>
</tr>
<tr>
<td>XX%KIND</td>
<td>type-param-inquiry</td>
</tr>
<tr>
<td>KIND</td>
<td>type-param-name</td>
</tr>
<tr>
<td>(S + T)</td>
<td>(expr)</td>
</tr>
</tbody>
</table>

ISO/IEC JTC 1/SC 22/WG5/N2118 149
10.1.2.3 Level-1 expressions

Defined unary operators have the highest operator precedence (Table 10.1). Level-1 expressions are primaries optionally operated on by defined unary operators:

R1002 level-1-expr is [ defined-unary-op ] primary
R1003 defined-unary-op is . letter [ letter ] ...

C1004 (R1003) A defined-unary-op shall not contain more than 63 letters and shall not be the same as any intrinsic-operator or logical-literal-constant.

NOTE 10.2
Simple examples of a level-1 expression are:

<table>
<thead>
<tr>
<th>Example</th>
<th>Syntactic class</th>
</tr>
</thead>
<tbody>
<tr>
<td>A .INVERSE. B</td>
<td>primary (R1001)</td>
</tr>
<tr>
<td></td>
<td>level-1-expr (R1002)</td>
</tr>
</tbody>
</table>

A more complicated example of a level-1 expression is:

. INVERSE. (A + B)

10.1.2.4 Level-2 expressions

Level-2 expressions are level-1 expressions optionally involving the numeric operators power-op, mult-op, and add-op.

R1004 mult-operand is level-1-expr [ power-op mult-operand ]
R1005 add-operand is [ add-operand mult-op ] mult-operand
R1006 level-2-expr is [ [ level-2-expr ] add-op ] add-operand
R1007 power-op is **
R1008 mult-op is * or /
R1009 add-op is + or −

NOTE 10.3
Simple examples of a level-2 expression are:

<table>
<thead>
<tr>
<th>Example</th>
<th>Syntactic class</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>level-1-expr</td>
<td>A is a primary. (R1002)</td>
</tr>
<tr>
<td>B ** C</td>
<td>mult-operand</td>
<td>B is a level-1-expr, ** is a power-op, and C is a mult-operand. (R1004)</td>
</tr>
<tr>
<td>D * E</td>
<td>add-operand</td>
<td>D is an add-operand, * is a mult-op, and E is a mult-operand. (R1005)</td>
</tr>
<tr>
<td>+1</td>
<td>level-2-expr</td>
<td>+ is an add-op and 1 is an add-operand. (R1006)</td>
</tr>
<tr>
<td>F - I</td>
<td>level-2-expr</td>
<td>F is a level-2-expr, – is an add-op, and I is an add-operand. (R1006)</td>
</tr>
</tbody>
</table>
NOTE 10.3 (cont.)

A more complicated example of a level-2 expression is:

\[- A + D * E + B ** C\]

10.1.2.5 Level-3 expressions

Level-3 expressions are level-2 expressions optionally involving the character operator `concat-op`.

R1010 \(\text{level-3-expr}\) is \([\text{level-3-expr} \ \text{concat-op} \ \text{level-2-expr}]\)

R1011 \(\text{concat-op}\) is `//`

NOTE 10.4

Simple examples of a level-3 expression are:

<table>
<thead>
<tr>
<th>Example</th>
<th>Syntactic class</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>(\text{level-2-expr}) (R1006)</td>
</tr>
<tr>
<td>B // C</td>
<td>(\text{level-3-expr}) (R1010)</td>
</tr>
</tbody>
</table>

A more complicated example of a level-3 expression is:

\(X \ // \ Y \ // \ 'ABCD'\)

10.1.2.6 Level-4 expressions

Level-4 expressions are level-3 expressions optionally involving the relational operators `rel-op`.

R1012 \(\text{level-4-expr}\) is \([\text{level-3-expr} \ \text{rel-op} \ \text{level-3-expr}]\)

R1013 \(\text{rel-op}\) is `==` or `/=` or `<` or `<=` or `>` or `>=`

NOTE 10.5

Simple examples of a level-4 expression are:

<table>
<thead>
<tr>
<th>Example</th>
<th>Syntactic class</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>(\text{level-3-expr}) (R1010)</td>
</tr>
<tr>
<td>B == C</td>
<td>(\text{level-4-expr}) (R1012)</td>
</tr>
<tr>
<td>D &lt; E</td>
<td>(\text{level-4-expr}) (R1012)</td>
</tr>
</tbody>
</table>

A more complicated example of a level-4 expression is:

\((A + B) \ /= \ C\)
10.1.2.7 Level-5 expressions

Level-5 expressions are level-4 expressions optionally involving the logical operators `not-op`, `and-op`, `or-op`, and `equiv-op`.

| R1014 | `and-operand` is `not-op level-4-expr` |
| R1015 | `or-operand` is `or-operand and-op and-operand` |
| R1016 | `equiv-operand` is `equiv-operand or-op or-operand` |
| R1017 | `level-5-expr` is `level-5-expr equiv-op equiv-operand` |
| R1018 | `not-op` is `.NOT.` |
| R1019 | `and-op` is `.AND.` |
| R1020 | `or-op` is `.OR.` |
| R1021 | `equiv-op` is `.EQV.` or `.NEQV.` |

NOTE 10.6

Simple examples of a level-5 expression are:

<table>
<thead>
<tr>
<th>Example</th>
<th>Syntactic class</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>A</code></td>
<td><code>level-4-expr</code> (R1012)</td>
</tr>
<tr>
<td><code>.NOT. B</code></td>
<td><code>and-operand</code> (R1014)</td>
</tr>
<tr>
<td><code>C .AND. D</code></td>
<td><code>or-operand</code> (R1015)</td>
</tr>
<tr>
<td><code>E .OR. F</code></td>
<td><code>equiv-operand</code> (R1016)</td>
</tr>
<tr>
<td><code>G .EQV. H</code></td>
<td><code>level-5-expr</code> (R1017)</td>
</tr>
<tr>
<td><code>S .NEQV. T</code></td>
<td><code>level-5-expr</code> (R1017)</td>
</tr>
</tbody>
</table>

A more complicated example of a level-5 expression is:

`A .AND. B .EQV. .NOT. C`

10.1.2.8 General form of an expression

Expressions are level-5 expressions optionally involving defined binary operators. Defined binary operators have the lowest operator precedence (Table 10.1).

| R1022 | `expr` is `expr defined-binary-op level-5-expr` |
| R1023 | `defined-binary-op` is `. letter [ letter ]` ... . |

C1005 (R1023) A `defined-binary-op` shall not contain more than 63 letters and shall not be the same as any `intrinsic-operator` or `logical-literal-constant`.

NOTE 10.7

Simple examples of an expression are:

<table>
<thead>
<tr>
<th>Example</th>
<th>Syntactic class</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>A</code></td>
<td><code>level-5-expr</code> (R1017)</td>
</tr>
<tr>
<td><code>B .UNION .C</code></td>
<td><code>expr</code> (R1022)</td>
</tr>
</tbody>
</table>
NOTE 10.7 (cont.)

More complicated examples of an expression are:

(B .INTERSECT. C) .UNION. (X - Y)
A + B == C * D
.INVERSE. (A + B)
A + B .AND. C * D
E // G == H (1:10)

10.1.3 Precedence of operators

There is a precedence among the intrinsic and extension operations corresponding to the form of expressions specified in 10.1.2, which determines the order in which the operands are combined unless the order is changed by the use of parentheses. This precedence order is summarized in Table 10.1.

<table>
<thead>
<tr>
<th>Category of operation</th>
<th>Operators</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extension</td>
<td>defined-unary-op</td>
<td>Highest</td>
</tr>
<tr>
<td>Numeric</td>
<td>**</td>
<td></td>
</tr>
<tr>
<td>Numeric</td>
<td>*, /</td>
<td></td>
</tr>
<tr>
<td>Numeric</td>
<td>unary +, –</td>
<td></td>
</tr>
<tr>
<td>Numeric</td>
<td>binary +, –</td>
<td></td>
</tr>
<tr>
<td>Character</td>
<td>//</td>
<td></td>
</tr>
<tr>
<td>Relational</td>
<td>.EQ., .NE., .LT., .LE., .GT., .GE., (==, /=, &lt;, &lt;=, &gt;, &gt;=)</td>
<td>(==, /=)</td>
</tr>
<tr>
<td>Logical</td>
<td>.NOT.</td>
<td>(==, /=)</td>
</tr>
<tr>
<td>Logical</td>
<td>.AND.</td>
<td>(==, /=)</td>
</tr>
<tr>
<td>Logical</td>
<td>.OR.</td>
<td>(==, /=)</td>
</tr>
<tr>
<td>Logical</td>
<td>.EQV., .NEQV.</td>
<td>(==, /=)</td>
</tr>
<tr>
<td>Extension</td>
<td>defined-binary-op</td>
<td>Lowest</td>
</tr>
</tbody>
</table>

The precedence of a defined operation is that of its operator.

NOTE 10.8

For example, in the expression

\[-A ** 2\]

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. The interpretation of the above expression is the same as the interpretation of the expression

\[-(A ** 2)\]

The general form of an expression (10.1.2) also establishes a precedence among operators in the same syntactic class. This precedence determines the order in which the operands are to be combined in determining the interpretation of the expression unless the order is changed by the use of parentheses.

NOTE 10.9

In interpreting a level-2-expr containing two or more binary operators + or –, each operand (add-operand) is combined from left to right. Similarly, the same left-to-right interpretation for a mult-operand in add-operand, as well as for other kinds of expressions, is a consequence of the general form. However, for interpreting a mult-operand expression when two or more exponentiation operators ** combine level-1-expr operands, each level-1-expr is combined from right to left.
NOTE 10.9 (cont.)

For example, the expressions

\[\begin{align*}
2.1 + 3.4 + 4.9 \\
2.1 * 3.4 * 4.9 \\
2.1 / 3.4 / 4.9 \\
2 ** 3 ** 4 \\
'AB' // 'CD' // 'EF'
\end{align*}\]

have the same interpretations as the expressions

\[\begin{align*}
(2.1 + 3.4) + 4.9 \\
(2.1 * 3.4) * 4.9 \\
(2.1 / 3.4) / 4.9 \\
2 ** (3 ** 4) \\
('AB' // 'CD') // 'EF'
\end{align*}\]

As a consequence of the general form (10.1.2), only the first `add-operand` of a `level-2-expr` can be preceded by the identity (+) or negation (–) operator. These formation rules do not permit expressions containing two consecutive numeric operators, such as A ** –B or A + –B. However, expressions such as A ** (–B) and A + (–B) are permitted. The rules do allow a binary operator or an intrinsic unary operator to be followed by a defined unary operator, such as:

\[\begin{align*}
A * .INVERSE. B \\
- .INVERSE. (B)
\end{align*}\]

As another example, in the expression

\[\begin{align*}
A .OR. B .AND. C
\end{align*}\]

the general form implies a higher precedence for the `.AND.` operator than for the `.OR.` operator; therefore, the interpretation of the above expression is the same as the interpretation of the expression

\[\begin{align*}
A .OR. (B .AND. C)
\end{align*}\]

NOTE 10.10

An expression can contain more than one category of operator. The logical expression

\[\begin{align*}
L .OR. A + B >= C
\end{align*}\]

where A, B, and C are of type real, and L is of type logical, contains a numeric operator, a relational operator, and a logical operator. This expression would be interpreted the same as the expression

\[\begin{align*}
L .OR. ((A + B) >= C)
\end{align*}\]

NOTE 10.11

If

- the operator ** is extended to type logical,
- the operator .STARSTAR. is defined to duplicate the function of ** on type real,
- .MINUS. is defined to duplicate the unary operator –, and
- L1 and L2 are type logical and X and Y are type real,

then in precedence: L1 ** L2 is higher than X * Y; X * Y is higher than X .STARSTAR. Y; and .MINUS. X is higher than –X.
10.1.4 Evaluation of operations

1 An intrinsic operation requires the values of its operands.

2 Execution of a function reference in the logical expression in an IF statement (11.1.8.4), the mask expression in a WHERE statement (10.2.3.1), or the concurrent-limits and concurrent-steps in a FORALL statement (10.2.4) is permitted to define variables in the subsidiary action-stmt, where-assignment-stmt, or forall-assignment-stmt respectively. Except in those cases:
   • the evaluation of a function reference shall neither affect nor be affected by the evaluation of any other entity within the statement;
   • if a function reference causes definition or undefinition of an actual argument of the function, that argument or any associated entities shall not appear elsewhere in the same statement.

NOTE 10.12
For example, the statements

\[
A(I) = F(I) \\
Y = G(X) + X
\]

are prohibited if the reference to F defines or undefines I or the reference to G defines or undefines X.

However, in the statements

\[
\text{IF } (F(X)) A = X \\
\text{WHERE } (G(X)) B = X
\]

the reference to F and/or the reference to G can define X.

3 The appearance of an array constructor requires the evaluation of each scalar-int-expr of the ac-implied-do-control in any ac-implied-do it may contain.

4 When an elemental binary operation is applied to a scalar and an array or to two arrays of the same shape, the operation is performed element-by-element on corresponding array elements of the array operands.

NOTE 10.13
For example, the array expression

\[
A + B
\]

produces an array of the same shape as A and B. The individual array elements of the result have the values of the first element of A added to the first element of B, the second element of A added to the second element of B, etc.

5 When an elemental unary operator operates on an array operand, the operation is performed element-by-element, and the result is the same shape as the operand. If an elemental operation is intrinsically pure or is implemented by a pure elemental function (15.8), the element operations may be performed simultaneously or in any order.

10.1.5 Intrinsic operations

10.1.5.1 Intrinsic operation classification

1 An intrinsic operation is either a unary or binary operation. An intrinsic unary operation is an operation of the form \( \text{intrinsic-operator } x \) where \( x \) is of an intrinsic type (7.4) listed in Table 10.2 for the unary intrinsic operator.

2 An intrinsic binary operation is an operation of the form \( x_1 \text{ intrinsic-operator } x_2 \) where \( x_1 \) and \( x_2 \) are conformable and of the intrinsic types (7.4) listed in Table 10.2 for the binary intrinsic operator.
A numeric intrinsic operation is an intrinsic operation for which the \textit{intrinsic-operator} is a numeric operator (+, –, *, /, or **). A numeric intrinsic operator is the operator in a numeric intrinsic operation.

The character intrinsic operation is the intrinsic operation for which the \textit{intrinsic-operator} is (//) and both operands are of type character with the same \texttt{kind} \texttt{type} \texttt{parameter}. The character intrinsic operator is the operator in a character intrinsic operation.

A logical intrinsic operation is an intrinsic operation for which the \textit{intrinsic-operator} is .\texttt{AND.}, .\texttt{OR.}, .\texttt{NOT.}, .\texttt{EQV.}, or .\texttt{NEQV.} and both operands are of type logical. A logical intrinsic operator is the operator in a logical intrinsic operation.

A relational intrinsic operator is an \textit{intrinsic-operator} that is .\texttt{EQ.}, .\texttt{NE.}, .\texttt{GT.}, .\texttt{GE.}, .\texttt{LT.}, .\texttt{LE.}, ==, \textsc{/=}, >, \textsc{>=}, \texttt{<}, or \texttt{<=}. A relational intrinsic operation is an intrinsic operation for which the \textit{intrinsic-operator} is a relational intrinsic operator. A numeric relational intrinsic operation is a relational intrinsic operation for which both operands are of \texttt{numeric type}. A character relational intrinsic operation is a relational intrinsic operation for which both operands are of type character. The \texttt{kind type parameters} of the operands of a character relational intrinsic operation shall be the same.

The interpretations defined in subclause 10.1.5 apply to both scalars and arrays; the interpretation for arrays is obtained by applying the interpretation for scalars element by element.

Table 10.2: \texttt{Type of operands and results for intrinsic operators}

<table>
<thead>
<tr>
<th>Intrinsic operator</th>
<th>Type of (x_1)</th>
<th>Type of (x_2)</th>
<th>Type of ([x_1] \ op \ x_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>//</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>.GT., .GE., .LT., .LE.</td>
<td>C, C</td>
<td>C</td>
<td>L</td>
</tr>
<tr>
<td>.NOT.</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>.\texttt{AND.}, .\texttt{OR.}, .\texttt{EQV.}, .\texttt{NEQV.}</td>
<td>L, L</td>
<td>L</td>
<td>L</td>
</tr>
</tbody>
</table>

\texttt{Note: The symbols I, R, Z, C, and L stand for the types integer, real, complex, character, and logical, respectively. Where more than one type for \(x_2\) is given, the type of the result of the operation is given in the same relative position in the next column.}

\texttt{NOTE 10.14 For example, if X is of type real and J is of type integer, the expression X + J is of type real.}

10.1.5.2 \texttt{Numeric intrinsic operations}

10.1.5.2.1 \texttt{Interpretation of numeric intrinsic operations}

The two operands of numeric intrinsic binary operations may be of different \texttt{numeric types} or different \texttt{kind type parameters}. Except for a value raised to an integer power, if the operands have different types or \texttt{kind type parameters}, the effect is as if each operand that differs in type or \texttt{kind type parameter} from those of the result is converted to the type and \texttt{kind} \texttt{type parameter} of the result before the operation is performed. When a value of type real or complex is raised to an integer power, the integer operand need not be converted.
A numeric operation is used to express a numeric computation. Evaluation of a numeric operation produces a numeric value. The permitted data types for operands of the numeric intrinsic operations are specified in 10.1.5.1.

The numeric operators and their interpretation in an expression are given in Table 10.3, where $x_1$ denotes the operand to the left of the operator and $x_2$ denotes the operand to the right of the operator.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Representing</th>
<th>Use of operator</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>Exponentiation</td>
<td>$x_1$ ** $x_2$</td>
<td>Raise $x_1$ to the power $x_2$</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>$x_1$ / $x_2$</td>
<td>Divide $x_1$ by $x_2$</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>$x_1$ * $x_2$</td>
<td>Multiply $x_1$ by $x_2$</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>$x_1$ - $x_2$</td>
<td>Subtract $x_2$ from $x_1$</td>
</tr>
<tr>
<td>-</td>
<td>Negation</td>
<td>$-x_2$</td>
<td>Negate $x_2$</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
<td>$x_1$ + $x_2$</td>
<td>Add $x_1$ and $x_2$</td>
</tr>
<tr>
<td>+</td>
<td>Identity</td>
<td>$+x_2$</td>
<td>Same as $x_2$</td>
</tr>
</tbody>
</table>

The interpretation of a division operation depends on the types of the operands (10.1.5.2.2).

If $x_1$ and $x_2$ are of type integer and $x_2$ has a negative value, the interpretation of $x_1$ ** $x_2$ is the same as the interpretation of 1/(abs($x_2$)), which is subject to the rules of integer division (10.1.5.2.2).

**NOTE 10.15**
For example, 2 ** (-3) has the value of 1/(2 ** 3), which is zero.

10.1.5.2.2 Integer division

One operand of type integer may be divided by another operand of type integer. Although the mathematical quotient of two integers is not necessarily an integer, Table 10.2 specifies that an expression involving the division operator with two operands of type integer is interpreted as an expression of type integer. The result of such an operation is the integer closest to the mathematical quotient and between zero and the mathematical quotient inclusively.

**NOTE 10.16**
For example, the expression (-8) / 3 has the value (-2).

10.1.5.2.3 Complex exponentiation

In the case of a complex value raised to a complex power, the value of the operation $x_1$ ** $x_2$ is the principal value of $x_1^{x_2}$.

10.1.5.2.4 Evaluation of numeric intrinsic operations

The execution of any numeric operation whose result is not defined by the arithmetic used by the processor is prohibited. Raising a negative real value to a real power is prohibited.

Once the interpretation of a numeric intrinsic operation is established, the processor may evaluate any mathematically equivalent expression, provided that the integrity of parentheses is not violated.

Two expressions of a numeric type are mathematically equivalent if, for all possible values of their primaries, their mathematical values are equal. However, mathematically equivalent expressions of numeric type may produce different computational results.
NOTE 10.17
Any difference between the values of the expressions \((1./3.)*3.\) and \(1.\) is a computational difference, not a mathematical difference. The difference between the values of the expressions \(5/2\) and \(5./2.\) is a mathematical difference, not a computational difference.

The mathematical definition of integer division is given in 10.1.5.2.2.

NOTE 10.18

The following are examples of expressions with allowable alternative forms that can be used by the processor in the evaluation of those expressions. \(A, B,\) and \(C\) represent arbitrary real or complex operands; \(I\) and \(J\) represent arbitrary integer operands; and \(X, Y,\) and \(Z\) represent arbitrary operands of numeric type.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Allowable alternative form</th>
</tr>
</thead>
<tbody>
<tr>
<td>(X + Y)</td>
<td>(Y + X)</td>
</tr>
<tr>
<td>(X * Y)</td>
<td>(Y * X)</td>
</tr>
<tr>
<td>(-X + Y)</td>
<td>(Y - X)</td>
</tr>
<tr>
<td>(X + Y + Z)</td>
<td>(X + (Y + Z))</td>
</tr>
<tr>
<td>(X - Y + Z)</td>
<td>(X - (Y - Z))</td>
</tr>
<tr>
<td>(X * A / Z)</td>
<td>(X * (A / Z))</td>
</tr>
<tr>
<td>(X * Y - X * Z)</td>
<td>(X * (Y - Z))</td>
</tr>
<tr>
<td>(A / B / C)</td>
<td>(A / (B * C))</td>
</tr>
<tr>
<td>(A / 5.0)</td>
<td>(0.2 * A)</td>
</tr>
</tbody>
</table>

The following are examples of expressions with forbidden alternative forms that cannot be used by a processor in the evaluation of those expressions.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Forbidden alternative form</th>
</tr>
</thead>
<tbody>
<tr>
<td>(I / 2)</td>
<td>(0.5 * I)</td>
</tr>
<tr>
<td>(X * I / J)</td>
<td>(X * (I / J))</td>
</tr>
<tr>
<td>(I / J / A)</td>
<td>(I / (J * A))</td>
</tr>
<tr>
<td>((X + Y) + Z)</td>
<td>(X + (Y + Z))</td>
</tr>
<tr>
<td>((X * Y) - (X * Z))</td>
<td>(X * (Y - Z))</td>
</tr>
<tr>
<td>(X * (Y - Z))</td>
<td>(X * Y - X * Z)</td>
</tr>
</tbody>
</table>

NOTE 10.19

In addition to the parentheses required to establish the desired interpretation, parentheses can be included to restrict the alternative forms that can be used by the processor in the actual evaluation of the expression. This is useful for controlling the magnitude and accuracy of intermediate values developed during the evaluation of an expression.

For example, in the expression

\[
A + (B - C)
\]

the parenthesized expression \((B - C)\) is evaluated and then added to \(A\).

The inclusion of parentheses could change the mathematical value of an expression. For example, the two expressions

\[
A * I \div J \\
A * (I \div J)
\]

could have different mathematical values if \(I\) and \(J\) are of type integer.
NOTE 10.20
Each operand in a numeric intrinsic operation has a type that can depend on the order of evaluation used by the processor.

For example, in the evaluation of the expression

\[ Z + R + I \]

where Z, R, and I represent data objects of complex, real, and integer type, respectively, the type of the operand that is added to I could be either complex or real, depending on which pair of operands (Z and R, R and I, or Z and I) is added first.

10.1.5.3 Character intrinsic operation

10.1.5.3.1 Interpretation of the character intrinsic operation

1 The character intrinsic operator `//` is used to concatenate two operands of type character with the same kind type parameter. Evaluation of the character intrinsic operation produces a result of type character.

2 The interpretation of the character intrinsic operator `//` when used to form an expression is given in Table 10.4, where \( x_1 \) denotes the operand to the left of the operator and \( x_2 \) denotes the operand to the right of the operator.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Representing Use of operator</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>//</code></td>
<td>Concatenation ( x_1 // x_2 )</td>
<td>Concatenate ( x_1 ) with ( x_2 )</td>
</tr>
</tbody>
</table>

3 The result of the character intrinsic operation \( x_1 // x_2 \) is a character string whose value is the value of \( x_1 \) concatenated on the right with the value of \( x_2 \) and whose length is the sum of the lengths of \( x_1 \) and \( x_2 \). Parentheses used to specify the order of evaluation have no effect on the value of a character expression.

NOTE 10.21
For example, the value of ('AB' // 'CDE') // 'F' is the string 'ABCDEF'. Also, the value of 'AB' // ('CDE' // 'F') is the string 'ABCDEF'.

10.1.5.3.2 Evaluation of the character intrinsic operation

1 A processor is only required to evaluate as much of the character intrinsic operation as is required by the context in which the expression appears.

NOTE 10.22
For example, the statements

```fortran
CHARACTER (LEN = 2) C1, C2, C3, CF
C1 = C2 // CF (C3)
```

do not require the function CF to be evaluated, because only the value of C2 is needed to determine the value of C1 because C1 and C2 both have a length of 2.

10.1.5.4 Logical intrinsic operations

10.1.5.4.1 Interpretation of logical intrinsic operations

1 A logical operation is used to express a logical computation. Evaluation of a logical operation produces a result of type logical. The permitted types for operands of the logical intrinsic operations are specified in 10.1.5.1.
The logical operators and their interpretation when used to form an expression are given in Table 10.5, where $x_1$ denotes the operand to the left of the operator and $x_2$ denotes the operand to the right of the operator.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Representing</th>
<th>Use of operator</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>.NOT.</td>
<td>Logical negation</td>
<td>.NOT. $x_2$</td>
<td>True if $x_2$ is false</td>
</tr>
<tr>
<td>.AND.</td>
<td>Logical conjunction</td>
<td>$x_1$ .AND. $x_2$</td>
<td>True if $x_1$ and $x_2$ are both true</td>
</tr>
<tr>
<td>.OR.</td>
<td>Logical inclusive disjunction</td>
<td>$x_1$ .OR. $x_2$</td>
<td>True if $x_1$ and/or $x_2$ is true</td>
</tr>
<tr>
<td>.EQV.</td>
<td>Logical equivalence</td>
<td>$x_1$ .EQV. $x_2$</td>
<td>True if both $x_1$ and $x_2$ are true or both are false</td>
</tr>
<tr>
<td>.NEQV.</td>
<td>Logical nonequivalence</td>
<td>$x_1$ .NEQV. $x_2$</td>
<td>True if either $x_1$ or $x_2$ is true, but not both</td>
</tr>
</tbody>
</table>

The values of the logical intrinsic operations are shown in Table 10.6.

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>.NOT. $x_2$</th>
<th>$x_1$ .AND. $x_2$</th>
<th>$x_1$ .OR. $x_2$</th>
<th>$x_1$ .EQV. $x_2$</th>
<th>$x_1$ .NEQV. $x_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

10.1.5.4.2 Evaluation of logical intrinsic operations

Once the interpretation of a logical intrinsic operation is established, the processor may evaluate any other expression that is logically equivalent, provided that the integrity of parentheses in any expression is not violated.

**NOTE 10.23**

For example, for the variables L1, L2, and L3 of type logical, the processor could choose to evaluate the expression

$L1$ .AND. $L2$ .AND. $L3$

as

$L1$ .AND. (L2 .AND. L3)

Two expressions of type logical are logically equivalent if their values are equal for all possible values of their primaries.

10.1.5.5 Relational intrinsic operations

10.1.5.5.1 Interpretation of relational intrinsic operations

A relational intrinsic operation is used to compare values of two operands using the relational intrinsic operators LT, LE, GT, GE, EQ, NE, <, <=, >, >=, ==, and /=. The permitted types for operands of the relational intrinsic operators are specified in 10.1.5.1.

The operators <, <=, >, >=, ==, and /= always have the same interpretations as the operators LT, LE, GT, GE, EQ, and NE, respectively.

**NOTE 10.24**

As shown in Table 10.2, a relational intrinsic operator cannot be used to compare the value of an expression of a numeric type with one of type character or logical. Also, two operands of type logical cannot be
NOTE 10.24 (cont.)

compared, a complex operand can be compared with another numeric operand only when the operator is .EQ., .NE., ==, or /=, and two character operands cannot be compared unless they have the same kind type parameter value.

NOTE 10.25
The collating sequence depends partially on the processor; however, the result of the use of the operators .EQ., .NE., ==, and /= does not depend on the collating sequence.

For nondefault character kinds, the blank padding character is processor dependent.

10.1.5.5.2 Evaluation of relational intrinsic operations

Once the interpretation of a relational intrinsic operation is established, the processor may evaluate any other expression that is relationally equivalent, provided that the integrity of parentheses in any expression is not
Two relational intrinsic operations are relationally equivalent if their logical values are equal for all possible values of their primaries.

NOTE 10.26
Whether an operand of a relational intrinsic operation could be an IEEE NaN affects whether expressions are equivalent. For example, if x or y could be a NaN, the expressions

\[ \text{.NOT. (x .LT. y)} \quad \text{and} \quad \text{x .GE. y} \]

are not equivalent.

10.1.6 Defined operations

10.1.6.1 Definitions

1 A defined operation is either a unary operation or a binary operation. A unary defined operation is an operation that has the form `defined-unary-op x2` or `intrinsic-operator x2` and that is defined by a function and a generic interface (7.5.5, 15.4.3.4).

2 A function defines the unary operation `op x2` if

1. the function is specified with a `FUNCTION` (15.6.2.2) or `ENTRY` (15.6.2.6) statement that specifies one dummy argument `d2`,
2. either
   1. a generic interface (15.4.3.2) provides the function with a `generic-spec` of `OPERATOR (op)`, or
   2. there is a generic binding (7.5.5) in the declared type of `x2` with a `generic-spec` of `OPERATOR (op)` and there is a corresponding binding to the function in the dynamic type of `x2`,
3. the type of `d2` is compatible with the dynamic type of `x2`,
4. the type parameters, if any, of `d2` match the corresponding type parameters of `x2`, and
5. either
   1. the rank of `x2` matches that of `d2` or
   2. the function is elemental and there is no other function that defines the operation.

3 If `d2` is an array, the shape of `x2` shall match the shape of `d2`.

4 A binary defined operation is an operation that has the form `x1 defined-binary-op x2` or `x1 intrinsic-operator x2` and that is defined by a function and a generic interface.

5 A function defines the binary operation `x1 op x2` if

1. the function is specified with a `FUNCTION` (15.6.2.2) or `ENTRY` (15.6.2.6) statement that specifies two dummy arguments, `d1` and `d2`,
2. either
   1. a generic interface (15.4.3.2) provides the function with a `generic-spec` of `OPERATOR (op)`, or
   2. there is a generic binding (7.5.5) in the declared type of `x1` or `x2` with a `generic-spec` of `OPERATOR (op)` and there is a corresponding binding to the function in the dynamic type of `x1` or `x2`, respectively,
3. the types of `d1` and `d2` are compatible with the dynamic types of `x1` and `x2`, respectively,
4. the type parameters, if any, of `d1` and `d2` match the corresponding type parameters of `x1` and `x2`, respectively, and
5. either
the ranks of \( x_1 \) and \( x_2 \) match those of \( d_1 \) and \( d_2 \), respectively, or

(b) the function is elemental, \( x_1 \) and \( x_2 \) are conformable, and there is no other function that defines the operation.

If \( d_1 \) or \( d_2 \) is an array, the shapes of \( x_1 \) and \( x_2 \) shall match the shapes of \( d_1 \) and \( d_2 \), respectively.

**NOTE 10.27**

An intrinsic operator can be used as the operator in a defined operation. In such a case, the generic properties of the operator are extended.

---

**10.1.6.2 Interpretation of a defined operation**

1. The interpretation of a defined operation is provided by the function that defines the operation.

2. The operators \(<, <=, >, >=, ==, \) and \( /=\) always have the same interpretations as the operators \( .LT., .LE., .GT., .GE., .EQ., \) and \( .NE.\), respectively.

**10.1.6.3 Evaluation of a defined operation**

1. Once the interpretation of a defined operation is established, the processor may evaluate any other expression that is equivalent, provided that the integrity of parentheses is not violated.

2. Two expressions of derived type are equivalent if their values are equal for all possible values of their primaries.

**10.1.7 Evaluation of operands**

1. It is not necessary for a processor to evaluate all of the operands of an expression, or to evaluate entirely each operand, if the value of the expression can be determined otherwise.

**NOTE 10.28**

This principle is most often applicable to logical expressions, zero-sized arrays, and zero-length strings, but it applies to all expressions.

For example, in evaluating the expression

\[ X > Y \ .OR. \ L(2) \]

where \( X, Y, \) and \( Z \) are real and \( L \) is a function of type logical, the function reference \( L(Z) \) need not be evaluated if \( X \) is greater than \( Y \). Similarly, in the array expression

\[ W(Z) + A \]

where \( A \) is of size zero and \( W \) is a function, the function reference \( W(Z) \) need not be evaluated.

2. If a statement contains a function reference in a part of an expression that need not be evaluated, all entities that would have become defined in the execution of that reference become undefined at the completion of evaluation of the expression containing the function reference.

**NOTE 10.29**

In the examples in Note 10.28, if \( L \) or \( W \) defines its argument, evaluation of the expressions under the specified conditions causes \( Z \) to become undefined, no matter whether or not \( L(Z) \) or \( W(Z) \) is evaluated.

3. If a statement contains a function reference in a part of an expression that need not be evaluated, no invocation of that function in that part of the expression shall execute an image control statement other than CRITICAL or END CRITICAL.
NOTE 10.30
This restriction is intended to avoid inadvertent deadlock caused by optimization.

10.1.8 Integrity of parentheses

The rules for evaluation specified in subclause 10.1.5 state certain conditions under which a processor may evaluate an expression that is different from the one specified by applying the rules given in 10.1.2 and rules for interpretation specified in subclause 10.1.5. However, any expression in parentheses shall be treated as a data entity.

NOTE 10.31
For example, in evaluating the expression \( A + (B - C) \) where \( A, B, \) and \( C \) are of numeric types, the difference of \( B \) and \( C \) shall be evaluated before the addition operation is performed; the processor shall not evaluate the mathematically equivalent expression \((A + B) - C\).

10.1.9 Type, type parameters, and shape of an expression

10.1.9.1 General

The type, type parameters, and shape of an expression depend on the operators and on the types, type parameters, and shapes of the primaries used in the expression, and are determined recursively from the syntactic form of the expression. The type of an expression is one of the intrinsic types (7.4) or a derived type (7.5).

If an expression is a polymorphic primary or defined operation, the type parameters and the declared and dynamic types of the expression are the same as those of the primary or defined operation. Otherwise the type parameters and dynamic type of the expression are the same as its declared type and type parameters; they are referred to simply as the type and type parameters of the expression.

R1024 logical-expr is expr
C1006 (R1024) logical-expr shall be of type logical.

R1025 default-char-expr is expr
C1007 (R1025) default-char-expr shall be default character.

R1026 int-expr is expr
C1008 (R1026) int-expr shall be of type integer.

R1027 numeric-expr is expr
C1009 (R1027) numeric-expr shall be of type integer, real, or complex.

10.1.9.2 Type, type parameters, and shape of a primary

The type, type parameters, and shape of a primary are determined according to whether the primary is a literal constant, designator, array constructor, structure constructor, function reference, type parameter inquiry, type parameter name, or parenthesized expression. If a primary is a literal constant, its type, type parameters, and shape are those of the literal constant. If it is a structure constructor, it is scalar and its type and type parameters are as described in 7.5.10. If it is an array constructor, its type, type parameters, and shape are as described in 7.8. If it is a designator or function reference, its type, type parameters, and shape are those of the designator (8.2, 8.5) or the function reference (15.5.3), respectively. If the function reference is generic (15.4.3.2, 16.7) then its type, type parameters, and shape are those of the specific function referenced, which is determined by the declared types, type parameters, and ranks of its actual arguments as specified in 15.5.5.2. If it is a type parameter inquiry or type parameter name, it is a scalar integer with the kind of the type parameter.
If a primary is a parenthesized expression, its type, type parameters, and shape are those of the expression.

The associated target object is referenced if a pointer appears as

- a primary in an intrinsic or defined operation,
- the `expr` of a parenthesized primary, or
- the only primary on the right-hand side of an intrinsic assignment statement.

The type, type parameters, and shape of the primary are those of the target. If the pointer is not associated with a target, it may appear as a primary only as an actual argument in a reference to a procedure whose corresponding dummy argument is declared to be a pointer, as the target in a pointer assignment statement, or as explicitly permitted elsewhere in this document.

A disassociated array pointer or an unallocated allocatable array has no shape but does have rank. The type, type parameters, and rank of the result of the intrinsic function NULL (16.9.144) depend on context.

### 10.1.9.3 Type, type parameters, and shape of the result of an operation

The type of the result of an intrinsic operation \([x_1] \text{ op } x_2\) is specified by Table 10.2. The shape of the result of an intrinsic operation is the shape of \(x_2\) if `op` is unary or if \(x_1\) is scalar, and is the shape of \(x_1\) otherwise.

The type, type parameters, and shape of the result of a defined operation \([x_1] \text{ op } x_2\) are specified by the function defining the operation (10.1.6).

An expression of an intrinsic type has a kind type parameter. An expression of type character also has a character length parameter.

The type parameters of the result of an intrinsic operation are as follows.

- For an expression \(x_1 // x_2\) where `//` is the character intrinsic operator and \(x_1\) and \(x_2\) are of type character, the character length parameter is the sum of the lengths of the operands and the kind type parameter is the kind type parameter of \(x_1\), which shall be the same as the kind type parameter of \(x_2\).
- For an expression `op x_2` where `op` is an intrinsic unary operator and \(x_2\) is of type integer, real, complex, or logical, the kind type parameter of the expression is that of the operand.
- For an expression \(x_1 \text{ op } x_2\) where `op` is a numeric intrinsic binary operator with both operands of the same kind type parameters, the kind type parameter of the expression is identical to that of each operand. In the case where both operands are integer with different kind type parameters, the kind type parameter of the expression is that of the operand with the greater decimal exponent range if the decimal exponent ranges are different; if the decimal exponent ranges are the same, the kind type parameter of the expression is processor dependent, but it is the same as that of one of the operands. In the case where both operands are any of type real or complex with different kind type parameters, the kind type parameter of the expression is that of the operand with the greater decimal precision if the decimal precisions are different; if the decimal precisions are the same, the kind type parameter of the expression is processor dependent, but it is the same as that of one of the operands.
- For an expression \(x_1 \text{ op } x_2\) where `op` is a logical intrinsic binary operator with both operands of the same kind type parameter, the kind type parameter of the expression is identical to that of each operand. In the case where both operands are of type logical with different kind type parameters, the kind type parameter of the expression is processor dependent, but it is the same as that of one of the operands.
- For an expression \(x_1 \text{ op } x_2\) where `op` is a relational intrinsic operator, the kind type parameter of the expression is default logical.
10.1.10 Conformability rules for elemental operations

An elemental operation is an intrinsic operation or a defined operation for which the function is elemental (15.8).

For all elemental binary operations, the two operands shall be conformable. In the case where one is a scalar and the other an array, the scalar is treated as if it were an array of the same shape as the array operand with every element, if any, of the array equal to the value of the scalar.

10.1.11 Specification expression

A specification expression is an expression with limitations that make it suitable for use in specifications such as length type parameters (C704) and array bounds (R817, R818). A specification-expr shall be a constant expression unless it is in an interface body (15.3.2), the specification part of a subprogram or BLOCK construct, a derived type definition, or the declaration-type-spec of a FUNCTION statement (15.6.2.2).

R1028 specification-expr is scalar-int-expr

C1010 (R1028) The scalar-int-expr shall be a restricted expression.

A restricted expression is an expression in which each operation is intrinsic or defined by a specification function and each primary is

(1) a constant or subobject of a constant,
(2) an object designator with a base object that is a dummy argument that has neither the OPTIONAL nor the INTENT (OUT) attribute,
(3) an object designator with a base object that is in a common block,
(4) an object designator with a base object that is made accessible by use or host association,
(5) an array constructor where each element and each scalar-int-expr of each ac-implied-do-control is a restricted expression,
(6) a structure constructor where each component is a restricted expression,
(7) a specification inquiry where each designator or argument is
   (a) a restricted expression or
   (b) a variable that is not an optional dummy argument, and whose properties inquired about are not
      (i) dependent on the upper bound of the last dimension of an assumed-size array,
      (ii) deferred, or
      (iii) defined by an expression that is not a restricted expression,
(8) a specification inquiry that is a constant expression,
(9) a reference to the intrinsic function PRESENT,
(10) a reference to any other standard intrinsic function where each argument is a restricted expression,
(11) a reference to a transformational function from the intrinsic module IEEE_ARITHMETIC, IEEE_-EXCEPTIONS, or ISO_C_BINDING, where each argument is a restricted expression,
(12) a reference to a specification function where each argument is a restricted expression,
(13) a type parameter of the derived type being defined,
(14) an ac-do-variable within an array constructor where each scalar-int-expr of the corresponding ac-implied-do-control is a restricted expression, or
(15) a restricted expression enclosed in parentheses,

where each subscript, section subscript, substring starting point, substring ending point, and type parameter value is a restricted expression, and where any final subroutine that is invoked is pure.

A specification inquiry is a reference to

(1) an intrinsic inquiry function other than PRESENT,
(2) a type parameter inquiry (9.4.5),
(3) an inquiry function from the intrinsic modules IEEE_ARITHMETIC and IEEE_EXCEPTIONS (17.10),
(4) the function C_SIZEOF from the intrinsic module ISO_C_BINDING (18.2.3.7), or
(5) the COMPILER_VERSION or COMPILER_OPTIONS function from the intrinsic module ISO_FORTRAN_ENV (16.10.2.6, 16.10.2.7).

A function is a specification function if it is a pure function, is not a standard intrinsic function, is not an internal function, is not a statement function, and does not have a dummy procedure argument.

Evaluation of a specification expression shall not directly or indirectly cause a procedure defined by the subprogram in which it appears to be invoked.

**NOTE 10.32**

Specification functions are nonintrinsic functions that can be used in specification expressions to determine the attributes of data objects. The requirement that they be pure ensures that they cannot have side effects that could affect other objects being declared in the same specification-part. The requirement that they not be internal ensures that they cannot inquire, via host association, about other objects being declared in the same specification-part. The prohibition against recursion avoids the creation of a new instance of a procedure while construction of one is in progress.

A variable in a specification expression shall have its type and type parameters, if any, specified by a previous declaration in the same scoping unit, 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 shall confirm the implied type and type parameters.

If a specification expression includes a specification inquiry that depends on a type parameter, array bound, or cobound of an entity specified in the same specification-part, the type parameter, array bound, or cobound shall be specified in a prior specification of the specification-part. The prior specification may be to the left of the specification inquiry in the same statement, but shall not be within the same entity-decl. If a specification expression includes a reference to the value of an element of an array specified in the same specification-part, the array shall be completely specified in prior declarations.

A generic entity referenced in a specification expression in the specification-part of a scoping unit shall have no specific procedures defined in the scoping unit, or its host scoping unit, subsequent to the specification expression.

**NOTE 10.33**

The following are examples of specification expressions:

```
LBOUND (B, 1) + 5 ! B is an assumed-shape dummy array
M + LEN (C) ! M and C are dummy arguments
2 * PRECISION (A) ! A is a real variable made accessible
      ! by a USE statement
```

**10.1.12 Constant expression**

A constant expression is an expression with limitations that make it suitable for use as a kind type parameter, initializer, or named constant. It is an expression in which each operation is intrinsic, and each primary is

(1) a constant or subobject of a constant,
(2) an array constructor where each element and each scalar-int-expr of each ac-implied-do-control is a constant expression,
(3) a structure constructor where each component-spec corresponding to
   (a) an allocatable component is a reference to the intrinsic function NULL,
(b) a pointer component is an initialization target or a reference to the intrinsic function NULL, and
(c) any other component is a constant expression,
(d) a specification inquiry where each designator or argument is
   (a) a constant expression or
   (b) a variable whose properties inquired about are not
      (i) assumed,
      (ii) deferred, or
      (iii) defined by an expression that is not a constant expression,
(5) a reference to an elemental standard intrinsic function, where each argument is a constant expression,
(6) a reference to a transformational standard intrinsic function other than COMMAND_ARGUMENT_COUNT, NULL, NUM_IMAGES, THIS_IMAGE, or TRANSFER, where each argument is a constant expression,
(7) a reference to the intrinsic function NULL that does not have an argument with a type parameter that is assumed or is defined by an expression that is not a constant expression,
(8) a reference to the intrinsic function TRANSFER where each argument is a constant expression and each ultimate pointer component of the SOURCE argument is disassociated,
(9) a reference to a transformational function from the intrinsic module IEEE_ARITHMETIC or IEEE_EXCEPTIONS, where each argument is a constant expression,
(10) a previously declared kind type parameter of the derived type being defined,
(11) a data-i-do-variable within a data-implied-do,
(12) an ac-do-variable within an array constructor where each scalar-int-expr of the corresponding ac-implied-do-control is a constant expression, or
(13) a constant expression enclosed in parentheses,

and where each subscript, section subscript, substring starting point, substring ending point, and type parameter value is a constant expression.

R1029 constant-expr is expr
C1011 (R1029) constant-expr shall be a constant expression.
R1030 default-char-constant-expr is default-char-expr
C1012 (R1030) default-char-constant-expr shall be a constant expression.
R1031 int-constant-expr is int-expr
C1013 (R1031) int-constant-expr shall be a constant expression.

If a constant expression includes a specification inquiry that depends on a type parameter or an array bound of an entity specified in the same specification-part, the type parameter or array bound shall be specified in a prior specification of the specification-part. The prior specification may be to the left of the specification inquiry in the same statement, but shall not be within the same entity-decl unless the specification inquiry appears within an initialization.

A generic entity referenced in a constant expression in the specification-part of a scoping unit shall have no specific procedures defined in that scoping unit, or its host scoping unit, subsequent to the constant expression.

NOTE 10.34
The following are examples of constant expressions:

\[
-3 + 4
\]

ISO/IEC JTC 1/SC 22/WG5/N2118
NOTE 10.34 (cont.)

\[
\begin{align*}
\text{'AB'} & \hspace{1cm} \text{'CD'} \\
\text{('AB' // 'CD') // 'EF'} & \\
\text{SIZE (A)} & \\
\text{DIGITS (X) + 4} & \\
4.0 \ast \text{ATAN (1.0)} & \\
\text{CEILING (number_of_decimal_digits / LOG10 (RADIX (0.0)))}
\end{align*}
\]

where A is an explicit-shape array with constant bounds and X is default real.

10.2 Assignment

10.2.1 Assignment statement

10.2.1.1 General form

\[ \text{R1032 assignment-stmt is variable = expr} \]

C1014 (R1032) The variable shall not be a whole assumed-size array.

NOTE 10.35

Examples of an assignment statement are:

\[
\begin{align*}
\text{A} &= 3.5 + \text{X} \ast \text{Y} \\
\text{I} &= \text{INT} (\text{A})
\end{align*}
\]

An assignment-stmt shall meet the requirements of either a defined assignment statement or an intrinsic assignment statement.

10.2.1.2 Intrinsic assignment statement

An intrinsic assignment statement is an assignment statement that is not a defined assignment statement (10.2.1.4). In an intrinsic assignment statement,

1. if the variable is polymorphic it shall be allocatable and not a coarray,
2. if \textit{expr} is an array then the variable shall also be an array,
3. the variable and \textit{expr} shall be conformable unless the variable is an allocatable array that has the same rank as \textit{expr} and is not a coarray,
4. if the variable is polymorphic it shall be type compatible with \textit{expr}; otherwise the declared types of the variable and \textit{expr} shall conform as specified in Table 10.8,
5. if the variable is of type character and of ISO 10646, ASCII, or default character kind, \textit{expr} shall be of ISO 10646, ASCII, or default character kind,
6. otherwise if the variable is of type character \textit{expr} shall have the same kind type parameter,
7. if the variable is of derived type each kind type parameter of the variable shall have the same value as the corresponding kind type parameter of \textit{expr}, and
8. if the variable is of derived type each length type parameter of the variable shall have the same value as the corresponding type parameter of \textit{expr} unless the variable is allocatable, is not a coarray, and its corresponding type parameter is deferred.

<table>
<thead>
<tr>
<th>Type of the variable</th>
<th>Type of \textit{expr}</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>integer, real, complex</td>
</tr>
</tbody>
</table>
Type conformance for the intrinsic assignment statement

<table>
<thead>
<tr>
<th>Type of the variable</th>
<th>Type of expr</th>
</tr>
</thead>
<tbody>
<tr>
<td>real</td>
<td>integer, real, complex</td>
</tr>
<tr>
<td>complex</td>
<td>integer, real, complex</td>
</tr>
<tr>
<td>character</td>
<td>character</td>
</tr>
<tr>
<td>logical</td>
<td>logical</td>
</tr>
<tr>
<td>derived type</td>
<td>same derived type as the variable</td>
</tr>
</tbody>
</table>

1. If the variable in an intrinsic assignment statement is a coindexed object,
   - the variable shall not be polymorphic,
   - the variable shall not have an allocatable ultimate component,
   - the variable shall be conformable with expr, and
   - each deferred length type parameter of the variable shall have the same value as the corresponding type parameter of expr.

2. If the variable is a pointer, it shall be associated with a definable target such that the type, type parameters, and shape of the target and expr conform.

10.2.1.3 Interpretation of intrinsic assignments

1. Execution of an intrinsic assignment causes, in effect, the evaluation of the expression expr and all expressions within variable (10.1), the possible conversion of expr to the type and type parameters of the variable (Table 10.9), and the definition of the variable with the resulting value. The execution of the assignment shall have the same effect as if the evaluation of expr and the evaluation of all expressions in variable occurred before any portion of the variable is defined by the assignment. The evaluation of expressions within variable shall neither affect nor be affected by the evaluation of expr.

2. If the variable is a pointer, the value of expr is assigned to the target of the variable.

3. If the variable is an unallocated allocatable array, expr shall have the same rank. If the variable is an allocated allocatable variable, it is deallocated if expr is an array of different shape, any of the corresponding length type parameter values of the variable and expr differ, or the variable is polymorphic and the dynamic type of the variable and expr differ. If the variable is or becomes an unallocated allocatable variable, it is then allocated with
   - the same dynamic type as expr if the variable is polymorphic,
   - each deferred type parameter equal to the corresponding type parameter of expr,
   - the same bounds as before if the variable is an array and expr is scalar, and
   - the shape of expr with each lower bound equal to the corresponding element of LBOUND(expr) if expr is an array.

NOTE 10.36

For example, given the declaration

```
CHARACTER(:), ALLOCATABLE :: NAME
```

then after the assignment statement

```
NAME = 'Dr. '//FIRST_NAME//' '//SURNAME
```

NAME will have the length LEN(FIRST_NAME) + LEN(SURNAME) + 5, even if it had previously been unallocated, or allocated with a different length. However, the assignment statement

```
NAME(:) = 'Dr. '//FIRST_NAME//' '//SURNAME
```
NOTE 10.36 (cont.)
is only conforming if NAME is already allocated at the time of the assignment; the assigned value is truncated or blank padded to the previously allocated length of NAME.

Both variable and expr may contain references to any portion of the variable.

NOTE 10.37
For example, in the character intrinsic assignment statement:

```
STRING (2:5) = STRING (1:4)
```

the assignment of the first character of STRING to the second character does not affect the evaluation of STRING (1:4). If the value of STRING prior to the assignment was 'ABCDEF', the value following the assignment is 'AABCDF'.

If expr is a scalar and the variable is an array, the expr is treated as if it were an array of the same shape as the variable with every element of the array equal to the scalar value of expr.

If the variable is an array, the assignment is performed element-by-element on corresponding array elements of the variable and expr.

NOTE 10.38
For example, if A and B are arrays of the same shape, the array intrinsic assignment

```
A = B
```

assigns the corresponding elements of B to those of A; that is, the first element of B is assigned to the first element of A, the second element of B is assigned to the second element of A, etc.

If C is an allocatable array of rank 1, then

```
C = PACK (ARRAY, ARRAY>0)
```

will cause C to contain all the positive elements of ARRAY in array element order; if C is not allocated or is allocated with the wrong size, it will be re-allocated to be of the correct size to hold the result of PACK.

The processor may perform the element-by-element assignment in any order.

NOTE 10.39
For example, the following program segment results in the values of the elements of array X being reversed:

```
REAL X (10)
...
X (1:10) = X (10:1:-1)
```

For an intrinsic assignment statement where the variable is of numeric type, the expr can have a different numeric type or kind type parameter, in which case the value of expr is converted to the type and kind type parameter of the variable according to the rules of Table 10.9.

<table>
<thead>
<tr>
<th>Type of the variable</th>
<th>Value assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>INT (expr, KIND = KIND (variable))</td>
</tr>
<tr>
<td>real</td>
<td>REAL (expr, KIND = KIND (variable))</td>
</tr>
<tr>
<td>complex</td>
<td>CMPLX (expr, KIND = KIND (variable))</td>
</tr>
</tbody>
</table>
Numeric conversion and the assignment statement (cont.)

<table>
<thead>
<tr>
<th>Type of the variable</th>
<th>Value Assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Note: INT, REAL, CMPLX, and KIND are the generic names of functions defined in 16.9.</td>
<td></td>
</tr>
</tbody>
</table>

For an intrinsic assignment statement where the variable is of type logical, the `expr` can have a different kind type parameter, in which case the value of `expr` is converted to the kind type parameter of the variable.

For an intrinsic assignment statement where the variable is of type character, the `expr` can have a different character length parameter in which case the conversion of `expr` to the length of the variable is as follows.

1. If the length of the variable is less than that of `expr`, the value of `expr` is truncated from the right until it is the same length as the variable.
2. If the length of the variable is greater than that of `expr`, the value of `expr` is extended on the right with blanks until it is the same length as the variable.

For an intrinsic assignment statement where the variable is of type character, if `expr` has a different kind type parameter, each character `c` in `expr` is converted to the kind type parameter of the variable by ACHAR (IACHAR(`c`), KIND(variable)).

NOTE 10.40
For nondefault character kinds, the blank padding character is processor dependent. When assigning a character expression to a variable of a different kind, each character of the expression that is not representable in the kind of the variable is replaced by a processor-dependent character.

For an intrinsic assignment of the type C_PTR or C_FUNPTR from the intrinsic module ISO_C_BINDING, the variable becomes undefined if the variable and `expr` are not on the same image.

NOTE 10.41
An intrinsic assignment statement for a variable of type C_PTR or C_FUNPTR is not permitted to involve a coindexed object, see C914, which prevents inappropriate copying from one image to another. However, such copying can occur as an intrinsic assignment for a component in a derived-type assignment.

An intrinsic assignment where the variable is of derived type is performed as if each component of the variable were assigned from the corresponding component of `expr` using pointer assignment (10.2.2) for each pointer component, defined assignment for each nonpointer nonallocatable component of a type that has a type-bound defined assignment consistent with the component, intrinsic assignment for each other nonpointer nonallocatable component, and intrinsic assignment for each allocated coarray component. For unallocated coarray components, the corresponding component of the variable shall be unallocated. For a noncoarray allocatable component the following sequence of operations is applied.

1. If the component of the variable is allocated, it is deallocated.
2. If the component of the value of `expr` is allocated, the corresponding component of the variable is allocated with the same dynamic type and type parameters as the component of the value of `expr`. If it is an array, it is allocated with the same bounds. The value of the component of the value of `expr` is then assigned to the corresponding component of the variable using defined assignment if the declared type of the component has a type-bound defined assignment consistent with the component, and intrinsic assignment for the dynamic type of that component otherwise.

The processor may perform the component-by-component assignment in any order or by any means that has the same effect.
NOTE 10.42
For an example of a derived-type intrinsic assignment statement, if C and D are of the same derived type
with a pointer component P and nonpointer components S, T, U, and V of type integer, logical, character,
and another derived type, respectively, the intrinsic

\[ C = D \]

pointer assigns D%P to C%P. It assigns D%S to C%S, D%T to C%T, and D%U to C%U using intrinsic
assignment. It assigns D%V to C%V using defined assignment if objects of that type have a compatible
type-bound defined assignment, and intrinsic assignment otherwise.

NOTE 10.43
If an allocatable component of \( expr \) is unallocated, the corresponding component of the variable has an
allocation status of unallocated after execution of the assignment.

10.2.1.4 Defined assignment statement

1 A defined assignment statement is an assignment statement that is defined by a subroutine and a generic interface
(7.5.5, 15.4.3.4.3) that specifies \texttt{ASSIGNMENT (=)}.

2 A subroutine defines the defined assignment \( x_1 = x_2 \) if

(1) the subroutine is specified with a \texttt{SUBROUTINE} (15.6.2.3) or \texttt{ENTRY} (15.6.2.6) statement that specifies
two dummy arguments, \( d_1 \) and \( d_2 \),

(2) either

(a) a generic interface (15.4.3.2) provides the subroutine with a \texttt{generic-spec} of \texttt{ASSIGNMENT (=)},
or

(b) there is a generic binding (7.5.5) in the declared type of \( x_1 \) or \( x_2 \) with a \texttt{generic-spec} of
\texttt{ASSIGNMENT (=)} and there is a corresponding binding to the subroutine in the dynamic
type of \( x_1 \) or \( x_2 \), respectively,

(3) the types of \( d_1 \) and \( d_2 \) are compatible with the dynamic types of \( x_1 \) and \( x_2 \), respectively,

(4) the type parameters, if any, of \( d_1 \) and \( d_2 \) match the corresponding type parameters of \( x_1 \) and \( x_2 \),
respectively, and

(5) either

(a) the ranks of \( x_1 \) and \( x_2 \) match those of \( d_1 \) and \( d_2 \) or

(b) the subroutine is elemental, \( x_2 \) is scalar or has the same rank as \( x_1 \), and there is no other
subroutine that defines the assignment.

3 If \( d_1 \) or \( d_2 \) is an array, the shapes of \( x_1 \) and \( x_2 \) shall match the shapes of \( d_1 \) and \( d_2 \), respectively. If the subroutine
is elemental, \( x_2 \) shall be conformable with \( x_1 \).

10.2.1.5 Interpretation of defined assignment statements

1 The interpretation of a defined assignment is provided by the subroutine that defines it.

2 If the defined assignment is an elemental assignment and the variable in the assignment is an array, the defined
assignment is performed element-by-element, on corresponding elements of the variable and \( expr \). If \( expr \) is a
scalar, it is treated as if it were an array of the same shape as the variable with every element of the array equal
to the scalar value of \( expr \).

NOTE 10.44
The rules of defined assignment (15.4.3.4.3), procedure references (15.5), subroutine references (15.5.4), and
elemental subroutine arguments (15.8.3) ensure that the defined assignment has the same effect as if the
evaluation of all operations in $x_2$ and $x_1$ occurs before any portion of $x_1$ is defined. If an elemental assignment
is defined by a pure elemental subroutine, the element assignments can be performed simultaneously or in
any order.

10.2.2 Pointer assignment

10.2.2.1 General

1 Pointer assignment causes a pointer to become associated with a target or causes its pointer association status
to become disassociated or undefined. Any previous association between the pointer and a target is broken.

2 Pointer assignment for a pointer component of a structure may also take place by execution of a derived-type
intrinsic assignment statement (10.2.1.3).

10.2.2.2 Syntax of the pointer assignment statement

R1033 pointer-assignment-stmt is
- data-pointer-object [ (bounds-spec-list) ] = expr
- data-pointer-object (bounds-remapping-list) = expr
- proc-pointer-object = expr

R1034 data-pointer-object is
- variable-name
- scalar-variable % data-pointer-component-name

C1015 (R1033) If data-target is not unlimited polymorphic, data-pointer-object shall be type compatible (7.3.2.3)
with it and the corresponding kind type parameters shall be equal.

C1016 (R1033) If data-target is unlimited polymorphic, data-pointer-object shall be unlimited polymorphic, or
of a type with the BIND attribute or the SEQUENCE attribute.

C1017 (R1033) If bounds-spec-list is specified, the number of bounds-specs shall equal the rank of data-pointer-
object.

C1018 (R1033) If bounds-remapping-list is specified, the number of bounds-remappings shall equal the rank of data-pointer-
object.

C1019 (R1033) If bounds-remapping-list is not specified, the ranks of data-pointer-object and data-target shall
be the same.

C1020 (R1033) A coarray data-target shall have the VOLATILE attribute if and only if the data-pointer-object
has the VOLATILE attribute.

C1021 (R1034) A variable-name shall have the POINTER attribute.

C1022 (R1034) A scalar-variable shall be a data-ref.

C1023 (R1034) A data-pointer-component-name shall be the name of a component of scalar-variable that is a
data pointer.

C1024 (R1034) A data-pointer-object shall not be a coindexed object.

R1035 bounds-spec is lower-bound-exp :

R1036 bounds-remapping is lower-bound-exp : upper-bound-exp

R1037 data-target is expr

C1025 (R1037) The expr shall be a designator that designates a variable with either the TARGET or POINTER
attribute and is not an array section with a vector subscript, or it shall be a reference to a function that
returns a data pointer.

C1026  (R1037) A data-target shall not be a coindexed object.

NOTE 10.45  
A data pointer and its target are always on the same image. A coarray can be of a derived type with pointer or allocatable subcomponents. For example, if PTR is a pointer component, and Z%PTR on image P has been associated with a target by execution of an ALLOCATE statement or a pointer assignment on image P, Z[P]%PTR will be a reference to that target.

R1038  proc-pointer-object is proc-pointer-name
or proc-component-ref

R1039  proc-component-ref is scalar-variable % procedure-component-name

C1027  (R1039) The scalar-variable shall be a data-ref that is not a coindexed object.

C1028  (R1039) The procedure-component-name shall be the name of a procedure pointer component of the declared type of scalar-variable.

R1040  proc-target is expr
or procedure-name
or proc-component-ref

C1029  (R1040) An expr shall be a reference to a function whose result is a procedure pointer.

C1030  (R1040) A procedure-name shall be the name of an internal, module, or dummy procedure, a procedure pointer, a specific intrinsic function listed in Table 16.2, or an external procedure that is accessed by use or host association, referenced in the scoping unit as a procedure, or that has the EXTERNAL attribute.

C1031  (R1040) The proc-target shall not be a nonintrinsic elemental procedure.

1 In a pointer assignment statement, data-pointer-object or proc-pointer-object denotes the pointer object and data-target or proc-target denotes the pointer target.

2 For pointer assignment performed by a derived-type intrinsic assignment statement, the pointer object is the pointer component of the variable and the pointer target is the corresponding component of expr.

10.2.2.3  Data pointer assignment

1 If the pointer object is not polymorphic (7.3.2.3) and the pointer target is polymorphic with dynamic type that differs from its declared type, the assignment target is the ancestor component of the pointer target that has the type of the pointer object. Otherwise, the assignment target is the pointer target.

2 If the pointer target is not a pointer, the pointer object becomes pointer associated with the assignment target; if the pointer target is a pointer with a target that is not on the same image, the pointer association status of the pointer object becomes undefined. Otherwise, the pointer association status of the pointer object becomes that of the pointer target; if the pointer target is associated with an object, the pointer object becomes associated with the assignment target. If the pointer target is allocatable, it shall be allocated.

NOTE 10.46  
A pointer assignment statement is not permitted to involve a coindexed pointer or target, see C1024 and C1026. This prevents a pointer assignment statement from associating a pointer with a target on another image. If such an association would otherwise be implied, the association status of the pointer becomes undefined. For example, a derived-type intrinsic assignment where the variable and expr are on different images and the variable has an ultimate pointer component.

3 If the pointer object is polymorphic, it assumes the dynamic type of the pointer target. If the pointer object is
of a type with the **BIND attribute** or the **SEQUENCE attribute**, the **dynamic type** of the pointer target shall be that type.

4 If the pointer target is a **disassociated** pointer, all nondeferred type parameters of the **declared type** of the pointer object that correspond to nondeferred type parameters of the pointer target shall have the same values as the corresponding type parameters of the pointer target.

5 Otherwise, all nondeferred type parameters of the **declared type** of the pointer object shall have the same values as the corresponding type parameters of the pointer target.

6 If the pointer object has nondeferred type parameters that correspond to **deferred type parameters** of the pointer target, the pointer target shall not be a pointer with undefined association status.

7 If the pointer object has the **CONTIGUOUS attribute**, the pointer target shall be contiguous.

8 If the target of a pointer is a **coarray**, the pointer shall have the **VOLATILE attribute** if and only if the coarray has the **VOLATILE attribute**.

9 If **bounds-remapping-list** appears, it specifies the upper and lower **bounds** of each dimension of the pointer, and thus the **extents**; the pointer target shall be **simply contiguous** (9.5.4) or of **rank** one, and shall not be a **disassociated** or **undefined** pointer. The number of elements of the pointer target shall not be less than the number implied by the **bounds-remapping-list**. The elements of the pointer object are associated with those of the pointer target, in array element order; if the pointer target has more elements than specified for the pointer object, the remaining elements are not associated with the pointer object.

10 If no **bounds-remapping-list** appears, the extent of a dimension of the pointer object is the extent of the corresponding dimension of the pointer target. If **bounds-spec-list** appears, it specifies the lower bounds; otherwise, the lower bound of each dimension is the result of the intrinsic function **LBOUND** (16.9.109) applied to the corresponding dimension of the pointer target. The upper bound of each dimension is one less than the sum of the lower bound and the extent.

10.2.2.4 Procedure pointer assignment

1 If the pointer target is not a pointer, the pointer object becomes **pointer associated** with the pointer target. Otherwise, the **pointer association** status of the pointer object becomes that of the pointer target; if the pointer target is associated with a procedure, the pointer object becomes associated with the same procedure.

2 The **host instance** (15.6.2.4) of an associated **procedure pointer** is the **host instance** of its target.

3 If the pointer object has an **explicit interface**, its **characteristics** shall be the same as the pointer target except that the pointer target may be pure even if the pointer object is not pure and the pointer target may be an **elemental** intrinsic procedure even if the pointer object is not **elemental**.

4 If the **characteristics** of the pointer object or the pointer target are such that an **explicit interface** is required, both the pointer object and the pointer target shall have an **explicit interface**.

5 If the pointer object has an **implicit interface** and is explicitly typed or referenced as a function, the pointer target shall be a function. If the pointer object has an **implicit interface** and is referenced as a subroutine, the pointer target shall be a subroutine.

6 If the pointer object is a function with an **implicit interface**, the pointer target shall be a function with the same type; corresponding type parameters shall have the same value.

7 If **procedure-name** is a specific procedure name that is also a **generic name**, only the specific procedure is associated with the pointer object.

10.2.2.5 Examples
NOTE 10.47

The following are examples of pointer assignment statements. (See Note 15.14 for declarations of P and BESSEL.)

```
NEW_NODE % LEFT => CURRENT_NODE
SIMPLE_NAME => TARGET_STRUCTURE % SUBSTRUCT % COMPONENT
PTR => NULL ( )
ROW => MAT2D (N, :)
WINDOW => MAT2D (I-1:I+1, J-1:J+1)
POINTER_OBJECT => POINTER_FUNCTION (ARG_1, ARG_2)
EVERY_OTHER => VECTOR (1:N:2)
WINDOW2 (0:, 0:) => MAT2D (ML:MU, NL:NU)
```

! P is a procedure pointer and BESSEL is a procedure with a ! compatible interface.
P => BESSEL

! Likewise for a structure component.
STRUCT % COMPONENT => BESSEL

NOTE 10.48

It is possible to obtain different-rank views of parts of an object by specifying upper bounds in pointer assignment statements. This requires that the object be either rank one or contiguous. Consider the following example, in which a matrix is under consideration. The matrix is stored as a rank-one object in MYDATA because its diagonal is needed for some reason – the diagonal cannot be gotten as a single object from a rank-two representation. The matrix is represented as a rank-two view of MYDATA.

```
real, target :: MYDATA ( NR*NC ) ! An automatic array
real, pointer :: MATRIX ( :, : ) ! A rank-two view of MYDATA
real, pointer :: VIEW_DIAG ( : )
MATRIX( 1:NR, 1:NC ) => MYDATA ! The MATRIX view of the data
VIEW_DIAG => MYDATA( 1::NR+1 ) ! The diagonal of MATRIX
```

Rows, columns, or blocks of the matrix can be accessed as sections of MATRIX.

Rank remapping can be applied to CONTIGUOUS arrays, for example:

```
REAL, CONTIGUOUS, POINTER :: A(:)
REAL, CONTIGUOUS, TARGET :: B(:, :) ! Dummy argument
A(1:SIZE(B)) => B ! Linear view of a rank-2 array
```

10.2.3 Masked array assignment – WHERE

10.2.3.1 General form of the masked array assignment

A masked array assignment is either a WHERE statement or a WHERE construct. It is used to mask the evaluation of expressions and assignment of values in array assignment statements, according to the value of a logical array expression.

```
R1041  where-stmt is WHERE ( mask-expr ) where-assignment-stmt
R1042  where-construct is  where-construct-stmt
                    [ where-body-construct ] ...
                    [ masked-elsewhere-stmt
                    [ where-body-construct ] ... ] ...
                    [ elsewhere-stmt
```
[ where-body-construct ] ...
end-where-stmt

R1043 where-construct-stmt is [where-construct-name:] WHERE ( mask-expr )
R1044 where-body-construct is where-assignment-stmt
or where-stmt
or where-construct
R1045 where-assignment-stmt is assignment-stmt
R1046 mask-expr is logical-expr
R1047 masked-elsewhere-stmt is ELSEWHERE (mask-expr) [where-construct-name]
R1048 elsewhere-stmt is ELSEWHERE [where-construct-name]
R1049 end-where-stmt is END WHERE [where-construct-name]

C1032 (R1045) A where-assignment-stmt that is a defined assignment shall be elemental.
C1033 (R1042) If the where-construct-stmt is identified by a where-construct-name, the corresponding end-
where-stmt shall specify the same where-construct-name. If the where-construct-stmt is not identified by
a where-construct-name, the corresponding end-where-stmt shall not specify a where-construct-name. If
an elsewhere-stmt or a masked-elsewhere-stmt is identified by a where-construct-name, the corresponding
where-construct-stmt shall specify the same where-construct-name.

C1034 (R1044) A statement that is part of a where-body-construct shall not be a branch target statement.

NOTE 10.49
Examples of a masked array assignment are:

WHERE (TEMP > 100.0) TEMP = TEMP - REDUCE_TEMP
WHERE (PRESSURE <= 1.0)
  PRESSURE = PRESSURE + INC_PRESSURE
  TEMP = TEMP - 5.0
ELSEWHERE
  RAINING = .TRUE.
END WHERE

10.2.3.2 Interpretation of masked array assignments

1 When a WHERE statement or a where-construct-stmt is executed, a control mask is established. In addition,
when a WHERE construct statement is executed, a pending control mask is established. If the statement does
not appear as part of a where-body-construct, the mask-expr of the statement is evaluated, and the control mask is
established to be the value of mask-expr. The pending control mask is established to have the value .NOT. mask-
expr upon execution of a WHERE construct statement that does not appear as part of a where-body-construct.
The mask-expr is evaluated only once.

2 Each statement in a WHERE construct is executed in sequence.

3 Upon execution of a masked-elsewhere-stmt, the following actions take place in sequence.

   (1) The control mask m_c is established to have the value of the pending control mask.
   (2) The pending control mask is established to have the value m_c .AND. (.NOT. mask-expr).
The control mask $m_c$ is established to have the value $m_c . \text{AND. } \text{mask-expr}$.

The mask-expr is evaluated at most once.

Upon execution of an ELSEWHERE statement, the control mask is established to have the value of the pending control mask. No new pending control mask value is established.

Upon execution of an ENDWHERE statement, the control mask and pending control mask are established to have the values they had prior to the execution of the corresponding WHERE construct statement. Following the execution of a WHERE statement that appears as a where-body-construct, the control mask is established to have the value it had prior to the execution of the WHERE statement.

NOTE 10.50

The establishment of control masks and the pending control mask is illustrated with the following example:

```
WHERE(cond1)  ! Statement 1
  . . .
ELSEWHERE(cond2)  ! Statement 2
  . . .
ELSEWHERE  ! Statement 3
  . . .
END WHERE
```

Following execution of statement 1, the control mask has the value cond1 and the pending control mask has the value .NOT. cond1. Following execution of statement 2, the control mask has the value (.NOT. cond1) .AND. cond2 and the pending control mask has the value (.NOT. cond1) .AND. (.NOT. cond2). Following execution of statement 3, the control mask has the value (.NOT. cond1) .AND. (.NOT. cond2). The false condition values are propagated through the execution of the masked ELSEWHERE statement.

Upon execution of a WHERE construct statement that is part of a where-body-construct, the pending control mask is established to have the value $m_c . \text{AND. } \text{mask-expr}$. The control mask is then established to have the value $m_c . \text{AND. } \text{mask-expr}$. The mask-expr is evaluated at most once.

Upon execution of a WHERE statement that is part of a where-body-construct, the control mask is established to have the value $m_c . \text{AND. } \text{mask-expr}$. The pending control mask is not altered.

If a nonelemental function reference occurs in the expr or variable of a where-assignment-stmt or in a mask-expr, the function is evaluated without any masked control; that is, all of its argument expressions are fully evaluated and the function is fully evaluated. If the result is an array and the reference is not within the argument list of a nonelemental function, elements corresponding to true values in the control mask are selected for use in evaluating the expr, variable or mask-expr.

If an elemental operation or function reference occurs in the expr or variable of a where-assignment-stmt or in a mask-expr, and is not within the argument list of a nonelemental function reference, the operation is performed or the function is evaluated only for the elements corresponding to true values of the control mask.

If an array constructor appears in a where-assignment-stmt or in a mask-expr, the array constructor is evaluated without any masked control and then the where-assignment-stmt is executed or the mask-expr is evaluated.

When a where-assignment-stmt is executed, the values of expr that correspond to true values of the control mask are assigned to the corresponding elements of the variable.

The value of the control mask is established by the execution of a WHERE statement, a WHERE construct statement, an ELSEWHERE statement, a masked ELSEWHERE statement, or an ENDWHERE statement. Subsequent changes to the value of entities in a mask-expr have no effect on the value of the control mask. The execution of a function reference in the mask expression of a WHERE statement is permitted to affect entities in...
the assignment statement.

NOTE 10.51

Examples of function references in masked array assignments are:

```fortran
WHERE (A > 0.0)                ! LOG is invoked only for positive elements.
  A = LOG (A)
  A = A / SUM (LOG (A))        ! LOG is invoked for all elements
  ! because SUM is transformational
END WHERE
```

10.2.4 FORALL

10.2.4.1 Form of the FORALL Construct

1 The FORALL construct allows multiple assignments, masked array (WHERE) assignments, and nested FORALL constructs and statements to be controlled by a single concurrent-control-list and scalar-mask-expr.

```
R1050 forall-construct is forall-construct-stmt
    [forall-body-construct ] ...
    end-forall-stmt

R1051 forall-construct-stmt is [ forall-construct-name ; ] FORALL concurrent-header

R1052 forall-body-construct is forall-assignment-stmt
    or where-stmt
    or where-construct
    or forall-construct
    or forall-stmt

R1053 forall-assignment-stmt is assignment-stmt
    or pointer-assignment-stmt

R1054 end-forall-stmt is END FORALL [forall-construct-name ]
```

C1035 (R1054) If the forall-construct-stmt has a forall-construct-name, the end-forall-stmt shall have the same forall-construct-name. If the end-forall-stmt has a forall-construct-name, the forall-construct-stmt shall have the same forall-construct-name.

C1036 (R1052) A statement in a forall-body-construct shall not define an index-name of the forall-construct.

C1037 (R1052) Any procedure referenced in a forall-body-construct, including one referenced by a defined operation, assignment, or finalization, shall be a pure procedure.

C1038 (R1052) A forall-body-construct shall not be a branch target.

2 The scope and attributes of an index-name in a concurrent-header in a FORALL construct or statement are described in 19.4.

10.2.4.2 Execution of the FORALL construct

10.2.4.2.1 Execution stages

1 There are three stages in the execution of a FORALL construct:

1. determination of the values for index-name variables,
2. evaluation of the scalar-mask-expr, and
3. execution of the FORALL body constructs.

10.2.4.2.2 Determination of the values for index variables

1 The values of the index variables are determined as they are for the DO CONCURRENT statement (11.1.7.4.2).
10.2.4.2.3 Evaluation of the mask expression

The mask expression is evaluated as it is for the DO CONCURRENT statement (11.1.7.4.2).

10.2.4.2.4 Execution of the FORALL body constructs

The forall-body-constructs are executed in the order in which they appear. Each construct is executed for all active combinations of the index-name values with the following interpretation:

1. Execution of a forall-assignment-stmt that is an assignment-stmt causes the evaluation of expr and all expressions within variable for all active combinations of index-name values. These evaluations may be done in any order. After all these evaluations have been performed, each expr value is assigned to the corresponding variable. The assignments may occur in any order.

2. Execution of a forall-assignment-stmt that is a pointer-assignment-stmt causes the evaluation of all expressions within data-target and data-pointer-object or proc-target and proc-pointer-object, the determination of any pointers within data-pointer-object or proc-pointer-object, and the determination of the target for all active combinations of index-name values. These evaluations may be done in any order. After all these evaluations have been performed, each data-pointer-object or proc-pointer-object is associated with the corresponding target. These associations may occur in any order.

3. In a forall-assignment-stmt, a defined assignment subroutine shall not reference any variable that becomes defined by the statement.

4. Each statement in a where-construct (10.2.3) within a forall-construct is executed in sequence. When a where-stmt, where-construct-stmt or masked-elsewhere-stmt is executed, the statement’s mask-expr is evaluated for all active combinations of index-name values as determined by the outer forall-constructs, masked by any control mask corresponding to outer where-constructs. Any where-assignment-stmt is executed for all active combinations of index-name values, masked by the control mask in effect for the where-assignment-stmt.

5. Execution of a forall-stmt or forall-construct causes the evaluation of the concurrent-limit and concurrent-step expressions in the concurrent-control-list for all active combinations of the index-name values of the outer FORALL construct. The set of combinations of index-name values for the inner FORALL is the union of the sets defined by these limits and steps for each active combination of the outer index-name values; it also includes the outer index-name values. The scalar-mask-expr is then evaluated for all combinations of the index-name values of the inner construct to produce a set of active combinations for the inner construct. If there is no scalar-mask-expr, it is as if it appeared with the value true. Each statement in the inner FORALL is then executed for each active combination of the index-name values.

10.2.4.3 The FORALL statement

The FORALL statement allows a single assignment statement or pointer assignment statement to be controlled by a set of index values and an optional mask expression.

R1055 forall-stmt is FORALL concurrent-header forall-assignment-stmt

1. A FORALL statement is equivalent to a FORALL construct containing a single forall-body-construct that is a forall-assignment-stmt.

2. The scope of an index-name in a forall-stmt is the statement itself (19.4).

10.2.4.4 Restrictions on FORALL constructs and statements

1. A many-to-one assignment is more than one assignment to the same object, or association of more than one target with the same pointer, whether the object is referenced directly or indirectly through a pointer. A many-to-one assignment shall not occur within a single statement in a FORALL construct or statement. It is possible to assign or pointer-assign to the same object in different assignment or pointer assignment statements in a FORALL construct.
NOTE 10.53

The appearance of each index-name in the identification of the left-hand side of an assignment statement is helpful in eliminating many-to-one assignments, but it is not sufficient to guarantee there will be none. For example, the following is allowed

\[
\text{FORALL } (I = 1:10) \\
A \{\text{INDEX } (I)\} = B(I) \\
\text{END FORALL}
\]

if and only if INDEX(1:10) contains no repeated values.

1 2 Within the scope of a FORALL construct, a nested FORALL statement or FORALL construct shall not have the same index-name.

2 The concurrent-header expressions within a nested FORALL may depend on the values of outer index-name variables.
11 Execution control

11.1 Executable constructs containing blocks

11.1.1 Blocks

The following are executable constructs that contain blocks:

- ASSOCIATE construct;
- BLOCK construct;
- CHANGE TEAM construct;
- CRITICAL construct;
- DO construct;
- IF construct;
- SELECT CASE construct;
- SELECT RANK construct;
- SELECT TYPE construct.

R1101 block is [execution-part-construct] ...

Executable constructs may be used to control which blocks of a program are executed or how many times a block is executed. Blocks are always bounded by statements that are particular to the construct in which they are embedded.

NOTE 11.1
An example of a construct containing a block is:

IF (A > 0.0) THEN
    B = SQRT (A) ! These two statements
    C = LOG (A) ! form a block.
END IF

11.1.2 Rules governing blocks

11.1.2.1 Control flow in blocks

Transfer of control to the interior of a block from outside the block is prohibited, except for the return from a procedure invoked within the block. Transfers within a block and transfers from the interior of a block to outside the block may occur.

Subroutine and function references (15.5.3, 15.5.4) may appear in a block.

11.1.2.2 Execution of a block

Execution of a block begins with the execution of the first executable construct in the block.

Execution of the block is completed when

- execution of the last executable construct in the block completes without branching to a statement within the block,
- a branch (11.2) within the block that has a branch target outside the block occurs,
• a **RETURN** statement within the block is executed, or
• an **EXIT** statement or **CYCLE** statement statement that belongs to a construct that contains the block is executed.

**NOTE 11.2**
The action that takes place at the terminal boundary depends on the particular construct and on the block within that construct.

### 11.1.3 ASSOCIATE construct

#### 11.1.3.1 Purpose and form of the ASSOCIATE construct

The ASSOCIATE construct associates named entities with expressions or variables during the execution of its block. These named construct entities (19.4) are associating entities (19.5.1.6). The names are *associate names*.

<table>
<thead>
<tr>
<th>Rule Id</th>
<th>Rule Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1102</td>
<td><strong>associate-construct</strong> is <strong>associate-stmt</strong> block end-associate-stmt</td>
</tr>
<tr>
<td>R1103</td>
<td><strong>associate-stmt</strong> is [ <strong>associate-construct-name</strong> : ] ASSOCIATE (association-list)</td>
</tr>
<tr>
<td>R1104</td>
<td><strong>association</strong> is <strong>associate-name</strong> =&gt; <strong>selector</strong></td>
</tr>
<tr>
<td>R1105</td>
<td><strong>selector</strong> is <strong>expr</strong> or <strong>variable</strong></td>
</tr>
</tbody>
</table>

#### C1101 (R1104) If **selector** is not a **variable** or is a **variable** that has a vector subscript, neither **associate-name** nor any subobject thereof shall appear in a variable definition context (19.6.7).

#### C1102 (R1104) An **associate-name** shall not be the same as another **associate-name** in the same **associate-stmt**.

#### C1103 (R1105) **variable** shall not be a coindexed object.

#### C1104 (R1105) **expr** shall not be a variable.

#### C1105 (R1105) **expr** shall not be a **designator** of a procedure pointer or a function reference that returns a procedure pointer.

<table>
<thead>
<tr>
<th>Rule Id</th>
<th>Rule Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1106</td>
<td><strong>end-associate-stmt</strong> is END ASSOCIATE [ <strong>associate-construct-name</strong> ]</td>
</tr>
<tr>
<td>C1106</td>
<td>(R1106) If the <strong>associate-stmt</strong> of an <strong>associate-construct</strong> specifies an <strong>associate-construct-name</strong>, the corresponding <strong>end-associate-stmt</strong> shall specify the same <strong>associate-construct-name</strong>. If the <strong>associate-stmt</strong> of an <strong>associate-construct</strong> does not specify an <strong>associate-construct-name</strong>, the corresponding <strong>end-associate-stmt</strong> shall not specify an <strong>associate-construct-name</strong>.</td>
</tr>
</tbody>
</table>

#### 11.1.3.2 Execution of the ASSOCIATE construct

Execution of an ASSOCIATE construct causes evaluation of every expression within every **selector** that is a variable designator and evaluation of every other **selector**, followed by execution of its block. During execution of that block each **associate name** identifies an entity which is associated (19.5.1.6) with the corresponding selector. The associating entity assumes the declared type and type parameters of the selector. If and only if the selector is polymorphic, the associating entity is polymorphic.

The other attributes of the associating entity are described in 11.1.3.3.

It is permissible to branch to an **end-associate-stmt** only from within its ASSOCIATE construct.
### 11.1.3.3 Other attributes of associate names

1. **Within an** ASSOCIATE, CHANGE TEAM, or SELECT TYPE **construct,** each **associating entity** has the same **rank** as its **associated selector.** The lower bound of each dimension is the result of the intrinsic function **LBOUND** (16.9.109) applied to the corresponding dimension of **selector.** The upper bound of each dimension is one less than the sum of the lower bound and the extent. The **associating entity** does not have the **ALLOCATABLE** or **POINTER** attributes; it has the **TARGET** attribute if and only if the selector is a variable and has either the **TARGET** or **POINTER** attribute.

2. **Within an** ASSOCIATE, SELECT RANK, or SELECT TYPE **construct,** each **associating entity** has the same **corank** as its associated selector. If the selector is a coarray, the **cobounds** of each codimension of the **associating entity** are the same as those of the selector.

3. **Within a** CHANGE TEAM **construct,** the **associating entity** is a **coarray.** Its **corank** and **cobounds** are as specified in its **codimension-decl.**

4. **Within an** ASSOCIATE, CHANGE TEAM, SELECT RANK, or SELECT TYPE **construct,** the **associating entity** has the **ASYNCHRONOUS** or **VOLATILE** attribute if and only if the selector is a variable and has the attribute. If the **associating entity** is polymorphic, it assumes the **dynamic type** and **type parameter** values of the selector. If the selector has the **OPTIONAL** attribute, it shall be present. The **associating entity** is **contiguous** if and only if the selector is **contiguous.**

5. The **associating entity** itself is a variable, but if the selector is not a **definable** variable, the **associating entity** is not definable and shall not be **definable** or become **undefined.** If a selector is not permitted to appear in a **variable definition context** (19.6.7), neither the **associate name** nor any subobject thereof shall appear in a variable definition context.

### 11.1.3.4 Examples of the ASSOCIATE construct

NOTE 11.3

The following example illustrates an association with an expression.

```fortran
ASSOCIATE ( Z => EXP (-(X**2+Y**2)) * COS (THETA) )
    PRINT *, A+Z, A-Z
END ASSOCIATE
```

The following example illustrates an association with a derived-type variable.

```fortran
ASSOCIATE ( XC => AX%B(I,J)%C )
    XC%DV = XC%DV + PRODUCT (XC%EV(1:N))
END ASSOCIATE
```

The following example illustrates association with an array section.

```fortran
ASSOCIATE ( ARRAY => AX%B(I,:)%C )
    ARRAY(N)%EV = ARRAY(N-1)%EV
END ASSOCIATE
```

The following example illustrates multiple associations.

```fortran
ASSOCIATE ( W => RESULT(I,J)%W, ZX => AX%B(I,J)%D, ZY => AY%B(I,J)%D )
    W = ZX*X + ZY*Y
END ASSOCIATE
```

### 11.1.4 BLOCK construct

1. The **BLOCK** construct is an executable construct that may contain declarations.
R1107  block-construct  is  block-stmt
  [ block-specification-part ]
  block
  end-block-stmt

R1108  block-stmt  is  [ block-construct-name : ] BLOCK

R1109  block-specification-part  is  [ use-stmt ]...
  [ import-stmt ]...
  [ implicit-part ]
  [ declaration-construct ]...
  specification-construct

R1110  end-block-stmt  is  END BLOCK [ block-construct-name ]

C1107  (R1107) A block-specification-part shall not contain a COMMON, EQUIVALENCE, INTENT, NAMELIST, OPTIONAL, statement function, or VALUE statement.

C1108  (R1107) A SAVE statement in a BLOCK construct shall contain a saved-entity-list that does not specify a common-block-name.

C1109  (R1107) If the block-stmt of a block-construct specifies a block-construct-name, the corresponding end-block-stmt shall specify the same block-construct-name. If the block-stmt does not specify a block-construct-name, the corresponding end-block-stmt shall not specify a block-construct-name.

2 Except for the ASYNCHRONOUS and VOLATILE statements, specifications in a BLOCK construct declare construct entities whose scope is that of the BLOCK construct (19.4). The appearance of the name of an object that is not a construct entity in an ASYNCHRONOUS or VOLATILE statement in a BLOCK construct specifies that the object has the attribute within the construct even if it does not have the attribute outside the construct.

3 Execution of a BLOCK construct causes evaluation of the specification expressions within its specification part in a processor-dependent order, followed by execution of its block.

4 It is permissible to branch to an end-block-stmt only from within its BLOCK construct.

NOTE 11.4

The following is an example of a BLOCK construct.

IF (swapxy) THEN
 BLOCK
  REAL (KIND (x)) tmp
  tmp = x
  x = y
  y = tmp
 END BLOCK
 END IF

Actions on a variable local to a BLOCK construct do not affect any variable of the same name outside the construct. For example,

F = 254E-2
BLOCK
 REAL F
 F = 39.37
END BLOCK
! F is still equal to 254E-2.

A SAVE statement outside a BLOCK construct does not affect variables local to the BLOCK construct,
NOTE 11.4 (cont.)

because a SAVE statement affects variables in its scoping unit rather than in its inclusive scope. For example,

```fortran
SUBROUTINE S
  ...
  SAVE
  ...
  BLOCK
    REAL X ! Not saved.
    REAL,SAVE :: Y(100) ! SAVE attribute is allowed.
    Z = 3 ! Implicitly declared in S, thus saved.
  ...
END BLOCK
...
END SUBROUTINE
```

11.1.5 CHANGE TEAM construct

11.1.5.1 Purpose and form of the CHANGE TEAM construct

The CHANGE TEAM construct changes the current team. Named construct entities (19.4) can be associated (19.5.1.6) with coarrays in the containing scoping unit, in the same way as for the ASSOCIATE construct.

R1111 change-team-construct is change-team-stmt
  block
end-change-team-stmt

R1112 change-team-stmt is [ team-construct-name: ] CHANGE TEAM ( team-variable [,
  coarray-association-list [,
  sync-stat-list ] )

R1113 coarray-association is codimension-decl = selector

R1114 end-change-team-stmt is END TEAM [ ( [ sync-stat-list ] ) ] [ team-construct-name ]

R1115 team-variable is scalar-variable

C1110 A branch (11.2) within a CHANGE TEAM construct shall not have a branch target that is outside the construct.

C1111 A RETURN statement shall not appear within a CHANGE TEAM construct.

C1112 An exit-stmt or cycle-stmt within a CHANGE TEAM construct shall not belong to an outer construct.

C1113 If the change-team-stmt of a change-team-construct specifies a team-construct-name, the corresponding end-change-team-stmt shall specify the same team-construct-name. If the change-team-stmt of a change-team-construct does not specify a team-construct-name, the corresponding end-change-team-stmt shall not specify a team-construct-name.

C1114 In a change-team-stmt, a coarray-name in a codimension-decl shall not be the same as a selector, or another coarray-name, in that statement.

C1115 A team-variable shall be of type TEAM_TYPE from the intrinsic module ISO_FORTRAN_ENV.

C1116 No selector shall appear more than once in a given change-team-stmt.

C1117 A selector in a coarray-association shall be a named coarray.
Each coarray-name in a codimension-decl in the CHANGE TEAM statement is an associate name which is associated with the corresponding selector. Each associating entity assumes the type and type parameters of its selector; it is polymorphic if and only if the selector is polymorphic. The other attributes of the associating entities are described in 11.1.3.3.

11.1.5.2 Execution of a CHANGE TEAM construct

The values of the team-variables on the active images of the team shall be those of team variables defined by synchronized executions of the same FORM TEAM statement (11.6.9) on all active images of the current team. The current team for the statements of the CHANGE TEAM block is the team specified by the value of the team-variable. The current team is not changed by a redefinition of the team variable during execution of the CHANGE TEAM construct. A CHANGE TEAM construct completes execution by executing its END TEAM statement, which restores the current team to the original team that was current for the CHANGE TEAM statement.

Execution of a CHANGE TEAM construct causes evaluation of the expressions within each codimension-decl in the CHANGE TEAM statement, followed by execution of its block. Each selector shall be an established coarray when the CHANGE TEAM statement begins execution.

It is permissible to branch to an end-change-team-stmt only from within its CHANGE TEAM construct.

An allocatable coarray that was allocated immediately before executing a CHANGE TEAM statement shall not be deallocated during execution of the construct. An allocatable coarray that was unallocated immediately before executing a CHANGE TEAM statement, and which is allocated immediately before executing the corresponding END TEAM statement, is deallocated by the execution of the END TEAM statement.

All active images of the current team shall execute the same CHANGE TEAM statement.

Successful execution of a CHANGE TEAM statement performs an implicit synchronization of all images of the new team that is identified by team-variable. On each image of the new team, execution of the segment following the CHANGE TEAM statement is delayed until all other images of that team have executed the same statement the same number of times in the original team.

If the new team contains a failed image and no other error condition occurs, there is an implicit synchronization of all active images of the new team. On each active image of the new team, execution of the segment following the CHANGE TEAM statement is delayed until all other active images of that team have executed the same statement the same number of times in the original team.

When a CHANGE TEAM construct completes execution, there is an implicit synchronization of all active images in the new team. On each active image of the new team, execution of the segment following the END TEAM statement is delayed until all other active images of this team have executed the same construct the same number of times in this team.

NOTE 11.5

Deallocation of an allocatable coarray that was not allocated at the beginning of a CHANGE TEAM construct, but is allocated at the end of execution of the construct, occurs even for allocatable coarrays with the SAVE attribute.

NOTE 11.6

Execution of a CHANGE TEAM statement includes a synchronization of the executing image with the other images that will be in the same team after execution of the CHANGE TEAM statement. Synchronization of these images occurs again when the corresponding END TEAM statement is executed.

If it is desired to synchronize all of the images in the team that is current when CHANGE TEAM begins execution, a SYNC ALL statement can be inserted just before the CHANGE TEAM statement. If similar semantics are desired following the END TEAM statement, a SYNC ALL statement could immediately follow the END TEAM statement.
11.1.6 CRITICAL construct

A CRITICAL construct limits execution of a block to one image at a time.

R1116 critical-construct is critical-stmt
    block
    end-critical-stmt

R1117 critical-stmt is [ critical-construct-name : ] CRITICAL [ ( [ sync-stat-list ] ) ]

R1118 end-critical-stmt is END CRITICAL [ critical-construct-name ]

C1118 (R1116) If the critical-stmt of a critical-construct specifies a critical-construct-name, the corresponding end-critical-stmt shall specify the same critical-construct-name. If the critical-stmt of a critical-construct does not specify a critical-construct-name, the corresponding end-critical-stmt shall not specify a critical-construct-name.

C1119 (R1116) The block of a critical-construct shall not contain a RETURN statement or an image control statement.

C1120 A branch (11.2) within a CRITICAL construct shall not have a branch target that is outside the construct.

2 Execution of the CRITICAL construct is completed when execution of its block is completed, or the executing image fails (5.3.6). A procedure invoked, directly or indirectly, from a CRITICAL construct shall not execute an image control statement.

3 The processor shall ensure that once an image has commenced executing block, no other image shall commence executing block until this image has completed execution of the construct. The image shall not execute an image control statement during the execution of block. The sequence of executed statements is therefore a segment (11.6.2). If image M completes execution of the construct without failing and image T is the next to execute the construct, the segment on image M precedes the segment on image T. Otherwise, if image M completes execution of the construct by failing, and image T is the next to execute the construct, the previous segment on image M precedes the segment on image T.

4 The effect of a STAT= or ERRMSG= specifier in a CRITICAL statement is specified in subclause 11.6.11.

5 It is permissible to branch to an end-critical-stmt only from within its CRITICAL construct.

NOTE 11.7

If more than one image executes the block of a CRITICAL construct without failing, its execution by one image always either precedes or succeeds its execution by another nonfailed image. Typically no other statement ordering is needed. Consider the following example:

CRITICAL
END CRITICAL

The definition of GLOBAL_COUNTER[1] by a particular image will always precede the reference to the same variable by the next image to execute the block.

NOTE 11.8

The following example permits a large number of jobs to be shared among the images:

INTEGER :: NUM_JOBS[*], JOB
IF (THIS_IMAGE() == 1) READ(*,*) NUM_JOBS
SYNC ALL
11.1.7 DO construct

11.1.7.1 Purpose and form of the DO construct

1 The DO construct specifies the repeated execution of a sequence of executable constructs. Such a repeated
sequence is called a loop.

2 The number of iterations of a loop can be determined at the beginning of execution of the DO construct, or can
be left indefinite (“DO forever” or DO WHILE). The execution order of the iterations can be left indeterminate
(DO CONCURRENT); except in this case, the loop can be terminated immediately (11.1.7.4.5). An iteration of
the loop can be curtailed by executing a CYCLE statement (11.1.7.4.4).

3 There are three phases in the execution of a DO construct: initiation of the loop, execution of each iteration of
the loop, and termination of the loop.

4 The scope and attributes of an index-name in a concurrent-header (DO CONCURRENT) are described in 19.4.

11.1.7.2 Form of the DO construct

R1119 do-construct is do-stmt

block
end-do

R1120 do-stmt is nonlabel-do-stmt
or label-do-stmt

R1121 label-do-stmt is [ do-construct-name : ] DO label [ loop-control ]

R1122 nonlabel-do-stmt is [ do-construct-name : ] DO [ loop-control ]

R1123 loop-control is [ , ] do-variable = scalar-int-expr, scalar-int-expr
or [ , ] WHILE ( scalar-logical-expr )
or [ , ] CONCURRENT concurrent-header concurrent-locality

R1124 do-variable is scalar-int-variable-name

C1121 (R1124) The do-variable shall be a variable of type integer.

R1125 concurrent-header is ( [ integer-type-spec :: ] concurrent-control-list [ , scalar-mask-expr ] )

R1126 concurrent-control is index-name = concurrent-limit : concurrent-limit [ : concurrent-step ]

R1127 concurrent-limit is scalar-int-expr
concurrent-step is scalar-int-expr

concurrent-locality is [ locality-spec ]...

locality-spec is LOCAL ( variable-name-list )
or LOCAL_INIT ( variable-name-list )
or SHARED ( variable-name-list )
or DEFAULT ( NONE )

C1122 (R1125) Any procedure referenced in the scalar-mask-expr, including one referenced by a defined operation, shall be a pure procedure (15.7).

C1123 (R1126) The index-name shall be a named scalar variable of type integer.

C1124 (R1126) A concurrent-limit or concurrent-step in a concurrent-control shall not contain a reference to any index-name in the concurrent-control-list in which it appears.

C1125 A variable-name in a locality-spec shall be the name of a variable in the innermost executable construct or scoping unit that includes the DO CONCURRENT statement.

C1126 A variable-name in a locality-spec shall not be the same as an index-name in the concurrent-header of the same DO CONCURRENT statement.

C1127 The name of a variable shall not appear in more than one variable-name-list, or more than once in a variable-name-list, in a locality-spec.

C1128 A variable-name that appears in a LOCAL or LOCAL_INIT locality-spec shall not have the ALLOCATABLE, INTENT (IN), or OPTIONAL attribute, shall not be of finalizable type, shall not be a nonpointer polymorphic dummy argument, and shall not be a coarray or an assumed-size array. A variable-name that is not permitted to appear in a variable definition context shall not appear in a LOCAL or LOCAL_INIT locality-spec.

C1129 A variable that is referenced by the scalar-mask-expr of a concurrent-header or by any concurrent-limit or concurrent-step in that concurrent-header shall not appear in a LOCAL locality-spec in the same DO CONCURRENT statement.

C1130 If the locality-spec DEFAULT ( NONE ) appears in a DO CONCURRENT statement, a variable that appears in the block of the construct and is not an index-name of that construct shall have its locality explicitly specified.

end-do is end-do-stmt

end-do-stmt is END DO [ do-construct-name ]

C1131 (R1119) If the do-stmt of a do-construct specifies a do-construct-name, the corresponding end-do shall be an end-do-stmt specifying the same do-construct-name. If the do-stmt of a do-construct does not specify a do-construct-name, the corresponding end-do shall not specify a do-construct-name.

C1132 (R1119) If the do-stmt is a nonlabel-do-stmt, the corresponding end-do shall be an end-do-stmt.

C1133 (R1119) If the do-stmt is a label-do-stmt, the corresponding end-do shall be identified with the same label.

It is permissible to branch to an end-do only from within its DO construct.

11.1.7.3 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 it terminates (11.1.7.4.5).

### 11.1.7.4 Execution of a DO construct

#### 11.1.7.4.1 Loop initiation

1. When the DO statement is executed, the DO construct becomes active. If `loop-control` is `[ , ] do-variable = scalar-int-expr1, scalar-int-expr2 [ , scalar-int-expr3 ]`

   the following steps are performed in sequence.

   1. The initial parameter $m_1$, the terminal parameter $m_2$, and the incrementation parameter $m_3$ are of type integer with the same kind type parameter as the `do-variable`. Their values are established by evaluating `scalar-int-expr1`, `scalar-int-expr2`, and `scalar-int-expr3`, respectively, including, if necessary, conversion to the kind type parameter of the `do-variable` according to the rules for numeric conversion (Table 10.9). If `scalar-int-expr3` does not appear, $m_3$ has the value 1. The value of $m_3$ shall not be zero.

   2. The DO variable becomes defined with the value of the initial parameter $m_1$.

   3. The iteration count is established and is the value of the expression $(m_2 - m_1 + m_3)/m_3$, unless that value is negative, in which case the iteration count is 0.

### NOTE 11.9

The iteration count is zero whenever:

- $m_1 > m_2$ and $m_3 > 0$, or
- $m_1 < m_2$ and $m_3 < 0$.

2. If `loop-control` is omitted, no iteration count is calculated. The effect is as if a large positive iteration count, impossible to decrement to zero, were established. If `loop-control` is `[ , ] WHILE (scalar-logical-expr)`, the effect is as if `loop-control` were omitted and the following statement inserted as the first statement of the block:

   ```
   IF (.NOT. (scalar-logical-expr)) EXIT
   ```

3. For a DO CONCURRENT construct, the values of the index variables for the iterations of the construct are determined by the rules in 11.1.7.4.2.

4. At the completion of the execution of the DO statement, the execution cycle begins.

### 11.1.7.4.2 DO CONCURRENT loop control

1. The `concurrent-limit` and `concurrent-step` expressions in the `concurrent-control-list` are evaluated. These expressions may be evaluated in any order. The set of values that a particular `index-name` variable assumes is determined as follows.

   1. The lower bound $m_1$, the upper bound $m_2$, and the step $m_3$ are of type integer with the same kind type parameter as the `index-name`. Their values are established by evaluating the first `concurrent-limit`, the second `concurrent-limit`, and the `concurrent-step` expressions, respectively, including, if necessary, conversion to the kind type parameter of the `index-name` according to the rules for numeric conversion (Table 10.9). If `concurrent-step` does not appear, $m_3$ has the value 1. The value of $m_3$ shall not be zero.

   2. Let the value of $max$ be $(m_2 - m_1 + m_3)/m_3$. If $max \leq 0$ for some `index-name`, the execution of the construct is complete. Otherwise, the set of values for the `index-name` is $m_1 + (k - 1) \times m_3$ where $k = 1, 2, ..., max$.

2. The set of combinations of `index-name` values is the Cartesian product of the sets defined by each triplet specification. An `index-name` becomes defined when this set is evaluated.
The scalar-mask-expr, if any, is evaluated for each combination of index-name values. If there is no scalar-mask-expr, it is as if it appeared with the value true. The index-name variables may be primaries in the scalar-mask-expr.

The set of active combinations of index-name values is the subset of all possible combinations for which the scalar-mask-expr has the value true.

NOTE 11.10
The index-name variables can appear in the mask, for example

\[ \text{DO CONCURRENT (I=1:10, J=1:10, A(I) > 0.0 .AND. B(J) < 1.0)} \]

11.1.7.4.3 The execution cycle

1 The execution cycle of a DO construct that is not a DO CONCURRENT construct consists of the following steps performed in sequence repeatedly until termination.

1. (1) The iteration count, if any, is tested. If it is zero, the loop terminates and the DO construct becomes inactive. If loop-control is \([,] \text{WHILE expanded-logical-expr}\), the expanded-logical-expr is evaluated; if the value of this expression is false, the loop terminates and the DO construct becomes inactive.

2. (2) The block of the loop is executed.

3. (3) The iteration count, if any, is decremented by one. The DO variable, if any, is incremented by the value of the incrementation parameter \(m\).

2 Except for the incrementation of the DO variable that occurs in step (3), the DO variable shall neither be redefined nor become undefined while the DO construct is active.

3 The block of a DO CONCURRENT construct is executed for every active combination of the index-name values. Each execution of the block is an iteration. The executions may occur in any order.

11.1.7.4.4 CYCLE statement

1 Execution of a loop iteration can be curtailed by executing a CYCLE statement that belongs to the construct.

R1133 cycle-stmt is CYCLE [do-construct-name]

C1134 If a do-construct-name appears on a CYCLE statement, the CYCLE statement shall be within that do-construct; otherwise, it shall be within at least one do-construct.

C1135 A cycle-stmt shall not appear within a CHANGE TEAM, CRITICAL, or DO CONCURRENT construct if it belongs to an outer construct.

2 A CYCLE statement belongs to a particular DO construct. If the CYCLE statement contains a DO construct name, it belongs to that DO construct; otherwise, it belongs to the innermost DO construct in which it appears.

3 Execution of a CYCLE statement that belongs to a DO construct that is not a DO CONCURRENT construct causes immediate progression to step (3) of the execution cycle of the DO construct to which it belongs.

4 Execution of a CYCLE statement that belongs to a DO CONCURRENT construct completes execution of that iteration of the construct.

5 In a DO construct, a transfer of control to the end-do has the same effect as execution of a CYCLE statement belonging to that construct.

11.1.7.4.5 Loop termination

1 For a DO construct that is not a DO CONCURRENT construct, the loop terminates, and the DO construct becomes inactive, when any of the following occurs.
• The iteration count is determined to be zero or the scalar-logical-expr is false, when tested during step (1) of the above execution cycle.
• An EXIT statement that belongs to the DO construct is executed.
• An EXIT or CYCLE statement that belongs to an outer construct and is within the DO construct is executed.
• A branch occurs within the DO construct and the branch target statement is outside the construct.
• A RETURN statement within the DO construct is executed.

2 For a DO CONCURRENT construct, the loop terminates, and the DO construct becomes inactive when all of the iterations have completed execution.

3 When a DO construct becomes inactive, the DO variable, if any, of the DO construct retains its last defined value.

11.1.7.5 Additional semantics for DO CONCURRENT constructs

C1136 A RETURN statement shall not appear within a DO CONCURRENT construct.
C1137 An image control statement shall not appear within a DO CONCURRENT construct.
C1138 A branch (11.2) within a DO CONCURRENT construct shall not have a branch target that is outside the construct.
C1139 A reference to an impure procedure shall not appear within a DO CONCURRENT construct.
C1140 A reference to the procedure IEEE_GET_FLAG, IEEE_SET_HALTING_MODE, or IEEE_GET_HALTING_MODE from the intrinsic module IEEE_EXCEPTIONS, shall not appear within a DO CONCURRENT construct.

1 The locality of a variable that appears in a DO CONCURRENT construct is LOCAL, LOCAL_INIT, SHARED, or unspecified.

2 A variable that has LOCAL or LOCAL_INIT locality is a construct entity with the same type, type parameters, rank as the variable with the same name in the innermost executable construct or scoping unit that includes the DO CONCURRENT construct, and the outside variable is inaccessible by that name within the construct. The construct entity has the ASYNCHRONOUS, CONTIGUOUS, POINTER, TARGET, or VOLATILE attribute if and only if the outside variable has that attribute; it does not have the BIND, INTENT, PROTECTED, SAVE or VALUE attribute, even if the outside variable has that attribute. If it is not a pointer, it has the same bounds as the outside variable. At the beginning of execution of each iteration,
   • if a variable with LOCAL locality is a pointer it has undefined pointer association status, and otherwise it is undefined except for any subobjects that are default-initialized;
   • a variable with LOCAL_INIT locality has the pointer association status and definition status of the outside variable with that name; the outside variable shall not be an undefined pointer or a nonallocatable nonpointer variable that is undefined.
If a variable with LOCAL or LOCAL_INIT locality becomes an affector of a pending input/output operation, the operation shall have completed before the end of the iteration. If a variable with LOCAL or LOCAL_INIT locality has the TARGET attribute, a pointer associated with it during an iteration becomes undefined when execution of that iteration completes.

3 If a variable has SHARED locality, appearances of the variable within the DO CONCURRENT construct refer to the variable in the innermost executable construct or scoping unit that includes the DO CONCURRENT construct. If it is defined or becomes undefined during any iteration, it shall not be referenced, defined, or become undefined during any other iteration. If it is allocated or deallocated during an iteration it shall not have its allocation or association status, dynamic type, array bounds, shape, or a deferred type parameter value inquired about in any other iteration. A noncontiguous array with SHARED locality shall not be supplied as an actual argument corresponding to a contiguous INTENT (INOUT) dummy argument.
If a variable has unspecified locality,

- if it is referenced in an iteration it shall either be previously defined during that iteration, or shall not be defined or become undefined during any other iteration; if it is defined or becomes undefined by more than one iteration it becomes undefined when the loop terminates;
- if it is noncontiguous and is supplied as an actual argument corresponding to a contiguous INTENT (IN-OUT) dummy argument in an iteration, it shall either be previously defined in that iteration or shall not be defined in any other iteration;
- if it is a pointer and is used in an iteration other than as the pointer in pointer assignment, allocation, or nullification, it shall either be previously pointer associated during that iteration or shall not have its pointer association changed during any iteration;
- if it is a pointer whose pointer association is changed in more than one iteration, it has an association status of undefined when the construct terminates;
- if it is allocatable and is allocated in more than one iteration, it shall have an allocation status of unallocated at the end of every iteration;
- if it is allocatable and is referenced, defined, deallocated, or has its allocation status, dynamic type, or a deferred type parameter value inquired about, in any iteration, it shall either be previously allocated in that iteration or shall not be allocated or deallocated in any other iteration.

A DO CONCURRENT construct shall not contain an input/output statement that has an ADVANCE= specifier.

If data are written to a file record or position in one iteration, that record or position in that file shall not be read from or written to in a different iteration. If records are written to a file connected for sequential access by more than one iteration, the ordering of records written by different iterations is processor dependent.

NOTE 11.11
The restrictions on referencing variables defined in an iteration of a DO CONCURRENT construct apply to any procedure invoked within the loop.

NOTE 11.12
The restrictions on the statements in a DO CONCURRENT construct are designed to ensure there are no data dependencies between iterations of the loop. This permits code optimizations that might otherwise be difficult or impossible because they would depend on properties of the program not visible to the compiler.

11.1.7.6 Examples of DO constructs

NOTE 11.13
The following program fragment computes a tensor product of two arrays:

```fortran
DO I = 1, M
  DO J = 1, N
    C (I, J) = DOT_PRODUCT (A (I, J, :), B(:, I, J))
  END DO
END DO
```

NOTE 11.14
The following program fragment contains a DO construct that uses the WHILE form of loop-control. The loop will continue to execute until an end-of-file or input/output error is encountered, at which point the DO statement terminates the loop. When a negative value of X is read, the program skips immediately to the next READ statement, bypassing most of the block of the loop.

```fortran
READ (IUN, '(1X, G14.7)'), IOSTAT = IOS) X
DO WHILE (IOS == 0)
  IF (X >= 0.) THEN
```

ISO/IEC JTC 1/SC 22/WG5/N2118
NOTE 11.14 (cont.)

CALL SUBA (X)
CALL SUBB (X)
...
CALL SUBZ (X)
ENDIF
READ (IUN, '(1X, G14.7)', IOSTAT = IOS) X
END DO

NOTE 11.15

The following example behaves exactly the same as the one in Note 11.14. However, the READ statement has been moved to the interior of the loop, so that only one READ statement is needed. Also, a CYCLE statement has been used to avoid an extra level of IF nesting.

DO ! A "DO WHILE + 1/2" loop
   READ (IUN, '(1X, G14.7)', IOSTAT = IOS) X
   IF (IOS /= 0) EXIT
   IF (X < 0.) CYCLE
   CALL SUBA (X)
   CALL SUBB (X)
   ...
   CALL SUBZ (X)
END DO

NOTE 11.16

The following example illustrates a case in which the user knows that there are no repeated values in the index array IND. The DO CONCURRENT construct makes it easier for the processor to generate vector gather/scatter code, unroll the loop, or parallelize the code for this loop, potentially improving performance.

INTEGER :: A(N), IND(N)
DO CONCURRENT (I=1:M)
   A(IND(I)) = I
END DO

NOTE 11.17

The following code demonstrates the use of the LOCAL clause so that the X inside the DO CONCURRENT construct is a temporary variable, and will not affect the X outside the construct.

X = 1.0
DO CONCURRENT (I=1:10) LOCAL (X)
   IF (A (I) > 0) THEN
      X = SQRT (A (I))
      A (I) = A (I) - X**2
   END IF
   B (I) = B (I) - A (I)
END DO
PRINT *, X ! Always prints 1.0.

NOTE 11.18

Additional examples of DO constructs are in C.6.3.
11.1.8 IF construct and statement

11.1.8.1 Purpose and form of the IF construct

1 The IF construct selects for execution at most one of its constituent blocks. The selection is based on a sequence of logical expressions.

R1134 if-construct is if-then-stmt

block
[ else-if-stmt

block ] ...
[ else-stmt

block ]
end-if-stmt

R1135 if-then-stmt is [ if-construct-name : ] IF ( scalar-logical-exp ) THEN

R1136 else-if-stmt is ELSE IF ( scalar-logical-exp ) THEN [ if-construct-name ]

R1137 else-stmt is ELSE [ if-construct-name ]

R1138 end-if-stmt is END IF [ if-construct-name ]

C1141 (R1134) If the if-then-stmt of an if-construct specifies an if-construct-name, the corresponding end-if-stmt shall specify the same if-construct-name. If the if-then-stmt of an if-construct does not specify an if-construct-name, the corresponding end-if-stmt shall not specify an if-construct-name. If an else-if-stmt or else-stmt specifies an if-construct-name, the corresponding if-then-stmt shall specify the same if-construct-name.

11.1.8.2 Execution of an IF construct

1 At most one of the blocks in the IF construct is executed. If there is an ELSE statement in the construct, exactly one of the blocks in the construct is executed. The scalar logical expressions are evaluated in the order of their appearance in the construct until a true value is found or an ELSE statement or END IF statement is encountered. If a true value or an ELSE statement is found, the block immediately following is executed and this completes the execution of the construct. The scalar logical expressions in any remaining ELSE IF statements of the IF construct are not evaluated. If none of the evaluated expressions is true and there is no ELSE statement, the execution of the construct is completed without the execution of any block within the construct.

2 It is permissible to branch to an END IF statement only from within its IF construct. Execution of an END IF statement has no effect.

11.1.8.3 Examples of IF constructs

NOTE 11.19

IF (CVAR == ‘RESET’) THEN
  I = 0; J = 0; K = 0
END IF

PROOF_DONE: IF (PROP) THEN
  WRITE (3, ‘(‘QED’’)’)
  STOP
ELSE
  PROP = NEXTPROP
END IF PROOF_DONE

IF (A > 0) THEN
  B = C/A
NOTE 11.19 (cont.)

```
IF (B > 0) THEN
  D = 1.0
END IF
ELSE IF (C > 0) THEN
  B = A/C
  D = -1.0
ELSE
  B = ABS (MAX (A, C))
  D = 0
END IF
```
R1145  case-selector is ( case-value-range-list )
or DEFAULT

C1145  (R1140) No more than one of the selectors of one of the CASE statements shall be DEFAULT.

R1146  case-value-range is case-value
or case-value :
or : case-value
or case-value : case-value

R1147  case-value is scalar-constant-expr

C1146  (R1140) For a given case-construct, each case-value shall be of the same type as case-expr. For character type, the kind type parameters shall be the same; character length differences are allowed.

C1147  (R1140) A case-value-range using a colon shall not be used if case-expr is of type logical.

C1148  (R1140) For a given case-construct, there shall be no possible value of the case-expr that matches more than one case-value-range.

11.1.9.2 Execution of a SELECT CASE construct

1 The execution of the SELECT CASE statement causes the case expression to be evaluated. For a case value range list, a match occurs if the case expression value matches any of the case value ranges in the list. For a case expression with a value of c, a match is determined as follows.

   (1) If the case value range contains a single value v without a colon, a match occurs for type logical if the expression c .EQV. v is true, and a match occurs for type integer or character if the expression c == v is true.

   (2) If the case value range is of the form low : high, a match occurs if the expression low <= c .AND. c <= high is true.

   (3) If the case value range is of the form low :, a match occurs if the expression low <= c is true.

   (4) If the case value range is of the form :, high, a match occurs if the expression c <= high is true.

   (5) If no other selector matches and a DEFAULT selector appears, it matches the case index.

   (6) If no other selector matches and the DEFAULT selector does not appear, there is no match.

2 The block following the CASE statement containing the matching selector, if any, is executed. This completes execution of the construct.

3 It is permissible to branch to an end-select-stmt only from within its SELECT CASE construct.

11.1.9.3 Examples of SELECT CASE constructs

NOTE 11.21

An integer signum function:

INTEGER FUNCTION SIGNUM (N)
SELECT CASE (N)
CASE (:-1)
    SIGNUM = -1
CASE (0)
    SIGNUM = 0
CASE (1:)
    SIGNUM = 1
END SELECT
END
NOTE 11.22
A code fragment to check for balanced parentheses:

```fortran
CHARACTER (80) :: LINE

LEVEL = 0
SCAN_LINE: DO I = 1, 80
  CHECK_PARENS: SELECT CASE (LINE (I:I))
  CASE ('(')
    LEVEL = LEVEL + 1
  CASE (')')
    LEVEL = LEVEL - 1
    IF (LEVEL < 0) THEN
      PRINT *, 'UNEXPECTED RIGHT PARENTHESIS'
      EXIT SCAN_LINE
    END IF
  CASE DEFAULT
    ! Ignore all other characters
  END SELECT CHECK_PARENS
END DO SCAN_LINE
IF (LEVEL > 0) THEN
  PRINT *, 'MISSING RIGHT PARENTHESIS'
END IF
```

NOTE 11.23
The following three fragments are equivalent:

```fortran
IF (SILLY == 1) THEN
  CALL THIS
ELSE
  CALL THAT
END IF
SELECT CASE (SILLY == 1)
CASE (.TRUE.)
  CALL THIS
CASE (.FALSE.)
  CALL THAT
END SELECT
SELECT CASE (SILLY)
CASE DEFAULT
  CALL OTHER
CASE (1)
  CALL THIS
END SELECT
```

NOTE 11.24
A code fragment showing several selections of one block:

```fortran
SELECT CASE (N)
CASE (1, 3:5, 8) ! Selects 1, 3, 4, 5, 8
  CALL SUB
CASE DEFAULT
  CALL OTHER
END SELECT
```
11.1.10 SELECT RANK construct

11.1.10.1 Purpose and form of the SELECT RANK construct

1 The SELECT RANK construct selects for execution at most one of its constituent blocks. The selection is based on the rank of an assumed-rank variable. A name is associated with the variable (19.4, 19.5.1.6), in the same way as for the ASSOCIATE construct.

R1148 select-rank-construct is select-rank-stmt
   [ select-rank-case-stmt
   block ]...
   end-select-rank-stmt

R1149 select-rank-stmt is [ select-construct-name : ] SELECT RANK □
   [ ( [ associate-name => ] selector ) ]

C1149 The selector in a select-rank-stmt shall be the name of an assumed-rank array.

R1150 select-rank-case-stmt is RANK ( scalar-int-constant-expr ) [ select-construct-name ]
   or RANK ( * ) [ select-construct-name ]
   or RANK DEFAULT [ select-construct-name ]

C1150 A scalar-int-constant-expr in a select-rank-case-stmt shall be nonnegative and less than or equal to the maximum possible rank of selector.

C1151 For a given select-rank-construct, the same rank value shall not be specified in more than one select-rank-case-stmt.

C1152 For a given select-rank-construct, there shall be at most one RANK ( * ) select-rank-case-stmt and at most one RANK DEFAULT select-rank-case-stmt.

C1153 If select-construct-name appears on a select-rank-case-stmt the corresponding select-rank-stmt shall specify the same select-construct-name.

C1154 A SELECT RANK construct shall not have a select-rank-case-stmt that is RANK ( * ) if the selector has the ALLOCATABLE or POINTER attribute.

R1151 end-select-rank-stmt is END SELECT [ select-construct-name ]

C1155 If the select-rank-stmt of a select-rank-construct specifies a select-construct-name, the corresponding end-select-rank-stmt does not specify a select-construct-name.

The associate name of a SELECT RANK construct is the associate-name if specified; otherwise it is the name that constitutes the selector.

11.1.10.2 Execution of the SELECT RANK construct

1 A SELECT RANK construct selects at most one block to be executed. During execution of that block, the associate name identifies an entity which is associated (19.5.1.6) with the selector. A RANK ( * ) statement matches the selector if the selector is argument associated with an assumed-size array. A RANK ( scalar-int-constant-expr ) statement matches the selector if the selector has that rank and is not argument associated with an assumed-size array. A RANK DEFAULT statement matches the selector if no other select-rank-case-stmt of the construct matches the selector. If a select-rank-case-stmt matches the selector, the block following that statement is executed; otherwise, control is transferred to the end-select-rank-stmt.

2 It is permissible to branch to an end-select-rank-stmt only from within its SELECT RANK construct.
11.1.10.3 Attributes of a SELECT RANK associate name

Within the block following a RANK DEFAULT statement, the associating entity (19.5.5) is assumed-rank and has
exactly the same attributes as the selector. Within the block following a RANK (* ) statement, the associating
entity has rank 1 and is assumed-size, as if it were declared with DIMENSION(1:__). Within the block following
a RANK ( scalar-int-constant-exp ) statement, the associating entity has the specified rank; the lower bound of
each dimension is the result of the intrinsic function LBOUND (16.9.109) applied to the corresponding dimension
of the selector, and the upper bound of each dimension is the result of the intrinsic function UBOUND (16.9.196)
applied to the corresponding dimension of the selector.

The associating entity has the ALLOCATABLE, POINTER, or TARGET attribute if the selector has that
attribute. The other attributes of the associating entity are described in 11.1.3.3.

11.1.10.4 Examples of the SELECT RANK construct

This example shows how to use a SELECT RANK construct to process scalars and rank-2 arrays; anything else
will be rejected as an error.

```
SUBROUTINE process(x)
  REAL x(..)
  !
  SELECT RANK(x)
  RANK (0)
  x = 0
  RANK (2)
  IF (SIZE(x,2)>=2) x(:,2) = 2
  RANK DEFAULT
  Print *, 'I did not expect rank', RANK(y), 'shape', SHAPE(y)
  ERROR STOP 'process bad arg'
END SELECT
```

The following example shows how to process assumed-size arrays, including how to use sequence association if
you want to do multi-dimensional processing of an assumed-size array.

```
SELECT RANK (y => x)
RANK (*)
  IF (RANK(x)==2) THEN
    ! Special code for the rank two case.
    CALL sequence_assoc_2(y, LBOUND(x,1), UBOUND(x,1), LBOUND(x,2))
  ELSE
    ! We just do all the other ranks in array element order.
    i = 1
    DO
      IF (y(i)==0) Exit
      y(i) = -y(i)
      i = i + 1
    END DO
  END IF
END SELECT
```

SUBROUTINE sequence_assoc_2(a, lb1, ub1, lb2)
  REAL a(lb1:ub1,lb2:*)
  j = lb2
  outer: DO
DO i=lb1,ub1
  IF (a(i,j)==0) EXIT outer
  a(i,j) = a(i,j)**2
END DO
j = j + 1
IF (ANY(a(:,j))==0) EXIT
j = j + 1
END DO outer
END SUBROUTINE

11.1.11 SELECT TYPE construct

11.1.11.1 Purpose and form of the SELECT TYPE construct

The SELECT TYPE construct selects for execution at most one of its constituent blocks. The selection is based on the dynamic type of an expression. A name is associated with the expression or variable (19.4, 19.5.1.6), in the same way as for the ASSOCIATE construct.

R1152 select-type-construct is select-type-stmt
  [ type-guard-stmt block ] ...
end-select-type-stmt

R1153 select-type-stmt is [ select-construct-name : ] SELECT TYPE [ ]
  [ [ associate-name => ] selector ]

C1156 (R1153) If selector is not a named variable, associate-name => shall appear.

C1157 (R1153) If selector is not a variable or is a variable that has a vector subscript, neither associate-name nor any subobject thereof shall appear in a variable definition context (19.6.7).

C1158 (R1153) The selector in a select-type-stmt shall be polymorphic.

R1154 type-guard-stmt is TYPE IS ( type-spec ) [ select-construct-name ]
or CLASS IS ( derived-type-spec ) [ select-construct-name ]
or CLASS DEFAULT [ select-construct-name ]

C1159 (R1154) The type-spec or derived-type-spec shall specify that each length type parameter is assumed.

C1160 (R1154) The type-spec or derived-type-spec shall not specify a type with the BIND attribute or the SEQUENCE attribute.

C1161 (R1152) If selector is not unlimited polymorphic, each TYPE IS or CLASS IS type-guard-stmt shall specify an extension of the declared type of selector.

C1162 (R1152) For a given select-type-construct, the same type and kind type parameter values shall not be specified in more than one TYPE IS type-guard-stmt and shall not be specified in more than one CLASS IS type-guard-stmt.

C1163 (R1152) For a given select-type-construct, there shall be at most one CLASS DEFAULT type-guard-stmt.

R1155 end-select-type-stmt is END SELECT [ select-construct-name ]

C1164 (R1152) If the select-type-stmt of a select-type-construct specifies a select-construct-name, the corresponding end-select-type-stmt shall specify the same select-construct-name. If the select-type-stmt of a select-type-construct does not specify a select-construct-name, the corresponding end-select-type-stmt shall not specify a select-construct-name. If a type-guard-stmt specifies a select-construct-name, the corresponding select-type-stmt shall specify the same select-construct-name.
The associate name of a SELECT TYPE construct is the associate-name if specified; otherwise it is the name that constitutes the selector.

11.1.11.2 Execution of the SELECT TYPE construct

Execution of a SELECT TYPE construct causes evaluation of every expression within a selector that is a variable designator, or evaluation of a selector that is not a variable designator.

A SELECT TYPE construct selects at most one block to be executed. During execution of that block, the associate name identifies an entity which is associated (19.5.1.6) with the selector.

A TYPE IS type guard statement matches the selector if the dynamic type and kind type parameter values of the selector are the same as those specified by the statement. A CLASS IS type guard statement matches the selector if the dynamic type of the selector is an extension of the type specified by the statement and the kind type parameter values specified by the statement are the same as the corresponding type parameter values of the dynamic type of the selector.

The block to be executed is selected as follows.

(1) If a TYPE IS type guard statement matches the selector, the block following that statement is executed.
(2) Otherwise, if exactly one CLASS IS type guard statement matches the selector, the block following that statement is executed.
(3) Otherwise, if several CLASS IS type guard statements match the selector, one of these statements will inevitably specify a type that is an extension of all the types specified in the others; the block following that statement is executed.
(4) Otherwise, if there is a CLASS DEFAULT type guard statement, the block following that statement is executed.
(5) Otherwise, no block is executed.

NOTE 11.25
This algorithm does not examine the type guard statements in source text order when it looks for a match; it selects the most particular type guard when there are several potential matches.

Within the block following a TYPE IS type guard statement, the associating entity (19.5.5) is not polymorphic (7.3.2.3), has the type named in the type guard statement, and has the type parameter values of the selector.

Within the block following a CLASS IS type guard statement, the associating entity is polymorphic and has the declared type named in the type guard statement. The type parameter values of the associating entity are the corresponding type parameter values of the selector.

Within the block following a CLASS DEFAULT type guard statement, the associating entity is polymorphic and has the same declared type as the selector. The type parameter values of the associating entity are those of the declared type of the selector.

NOTE 11.26
If the declared type of the selector is T, specifying CLASS DEFAULT has the same effect as specifying CLASS IS (T).

The other attributes of the associating entity are described in 11.1.3.3.

It is permissible to branch to an end-select-type-stmt only from within its SELECT TYPE construct.

11.1.11.3 Examples of the SELECT TYPE construct
NOTE 11.27

```fortran
TYPE POINT
  REAL :: X, Y
END TYPE POINT

TYPE, EXTENDS(POINT) :: POINT_3D
  REAL :: Z
END TYPE POINT_3D

TYPE, EXTENDS(POINT) :: COLOR_POINT
  INTEGER :: COLOR
END TYPE COLOR_POINT

TYPE(POINT), TARGET :: P
TYPE(POINT_3D), TARGET :: P3
TYPE(COLOR_POINT), TARGET :: C
CLASS(POINT), POINTER :: P_OR_C
P_OR_C => C
SELECT TYPE ( A => P_OR_C )
CLASS IS ( POINT )
  ! "CLASS ( POINT ) :: A" implied here
  PRINT *, A%X, A%Y ! This block gets executed
TYPE IS ( POINT_3D )
  ! "TYPE ( POINT_3D ) :: A" implied here
  PRINT *, A%X, A%Y, A%Z
END SELECT
```

NOTE 11.28

The following example illustrates the omission of `associate-name`. It uses the declarations from Note 11.27.

```fortran
P_OR_C => P3
SELECT TYPE ( P_OR_C )
CLASS IS ( POINT )
  ! "CLASS ( POINT ) :: P_OR_C" implied here
  PRINT *, P_OR_C%X, P_OR_C%Y
TYPE IS ( POINT_3D )
  ! "TYPE ( POINT_3D ) :: P_OR_C" implied here
  PRINT *, P_OR_C%X, P_OR_C%Y, P_OR_C%Z ! This block gets executed
END SELECT
```

11.1.12 EXIT statement

1 The EXIT statement provides one way of terminating a loop, or completing execution of another construct.

R1156 `exit-stmt` is `EXIT [ construct-name ]`

C1165 If a `construct-name` appears on an EXIT statement, the EXIT statement shall be within that construct; otherwise, it shall be within at least one `do-construct`.

2 An EXIT statement belongs to a particular construct. If a construct name appears, the EXIT statement belongs to that construct; otherwise, it belongs to the innermost `DO construct` in which it appears.

C1166 An `exit-stmt` shall not appear within a `CHANGE TEAM`, `CRITICAL`, or `DO CONCURRENT` construct if it belongs to that construct or an outer construct.

3 When an EXIT statement that belongs to a `DO construct` is executed, it terminates the loop (11.1.7.4.5) and any active loops contained within the terminated loop. When an EXIT statement that belongs to a non-DO
construct is executed, it terminates any active loops contained within that construct, and completes execution of that construct.

11.2 Branching

11.2.1 Branch concepts

Branching is used to alter the normal execution sequence. A branch causes a transfer of control from one statement to a labeled branch target statement in the same inclusive scope. Branching may be caused by a GO TO statement, a computed GO TO statement, a CALL statement that has an `alt-return-spec`, or an input/output statement that has an `END=`, `EOR=`, or `ERR=` specifier. Although procedure references and control constructs can cause transfer of control, they are not branches. A branch target statement is an `action-stmt`, `associate-stmt`, `end-associate-stmt`, `if-then-stmt`, `end-if-stmt`, `select-case-stmt`, `end-select-case-stmt`, `select-rank-stmt`, `end-select-rank-stmt`, `select-type-stmt`, `end-select-type-stmt`, `do-stmt`, `end-do-stmt`, `block-stmt`, `end-block-stmt`, `critical-stmt`, `end-critical-stmt`, `forall-construct-stmt`, `forall-stmt`, `where-construct-stmt`, `end-function-stmt`, `end-mp-subprogram-stmt`, `end-program-stmt`, or `end-subroutine-stmt`.

11.2.2 GO TO statement

11.2.3 Computed GO TO statement

11.3 CONTINUE statement

11.4 STOP and ERROR STOP statements

Execution of a STOP statement initiates normal termination of execution. Execution of an ERROR STOP statement initiates error termination of execution.
When an image is terminated by a STOP or ERROR STOP statement, its stop code, if any, is made available in a processor-dependent manner. If the stop-code is an integer, it is recommended that the value be used as the process exit status, if the processor supports that concept. If the stop-code in a STOP statement is of type character or does not appear, or if an end-program-stmt is executed, it is recommended that the value zero be supplied as the process exit status, if the processor supports that concept. If the stop-code in an ERROR STOP statement is of type character or does not appear, or if an end-program-stmt is executed, it is recommended that a processor-dependent nonzero value be supplied as the process exit status, if the processor supports that concept.

3 If QUIET= is omitted or the scalar-logical-expr has the value false:
   • if any exception (17) is signaling on that image, the processor shall issue a warning indicating which exceptions are signaling, and this warning shall be on the unit identified by the named constant ERROR_UNIT from the intrinsic module ISO_FORTRAN_ENV (16.10.2.9);
   • if a stop code is specified, it is recommended that it be made available by formatted output to the same unit.

4 If QUIET= appears and the scalar-logical-expr has the value true, no output of signaling exceptions or stop code shall be produced.

NOTE 11.29
When normal termination occurs on more than one image, it is expected that a processor-dependent summary of any stop codes and signaling exceptions will be made available.

NOTE 11.30
If the integer stop-code is used as the process exit status, the processor might be able to interpret only values within a limited range, or only a limited portion of the integer value (for example, only the least-significant 8 bits).

11.5 FAIL IMAGE statement

R1163 fail-image-stmt is FAIL IMAGE

Execution of a FAIL IMAGE statement causes the executing image to cease participating in program execution without initiating termination. No further statements are executed by that image.

NOTE 11.31
The FAIL IMAGE statement allows a program to test a recovery algorithm without needing an actual failure.

On a processor that does not have the ability to detect that an image has failed, execution of a FAIL IMAGE statement might provide a simulated failure environment that provides debug information.

In a piece of code that executes about once a second, invoking this subroutine on an image

```
SUBROUTINE FAIL
   REAL :: X
   CALL RANDOM_NUMBER (X)
   IF (X<0.001) FAIL IMAGE
END SUBROUTINE FAIL
```

will cause that image to have an independent 1/1000 chance of failure every second if the random number generators on different images are independent.

Note that FAIL IMAGE is not an image control statement.
11.6 Image execution control

11.6.1 Image control statements

The execution sequence on each image is specified in 5.3.5.

Execution of an image control statement divides the execution sequence on an image into segments. Each of the following is an image control statement:

- SYNC ALL statement;
- SYNC IMAGES statement;
- SYNC MEMORY statement;
- SYNC TEAM statement;
- ALLOCATE or DEALLOCATE statement that has a coarray allocate-object;
- CHANGE TEAM or END TEAM statement (11.1.5);
- CRITICAL or END CRITICAL statement (11.1.6);
- EVENT POST or EVENT WAIT statement;
- FORM TEAM statement;
- LOCK or UNLOCK statement;
- any statement that completes execution of a block or procedure and which results in the implicit deallocation of a coarray;
- a CALL statement that references the intrinsic subroutine MOVE_ALLOC with coarray arguments;
- STOP statement;
- END statement of a main program.

All image control statements except CRITICAL, END CRITICAL, EVENT POST, EVENT WAIT, FORM TEAM, LOCK, and UNLOCK include the effect of executing a SYNC MEMORY statement (11.6.5).

During an execution of a statement that invokes more than one procedure, at most one invocation shall cause execution of an image control statement other than CRITICAL or END CRITICAL.

11.6.2 Segments

On each image, the sequence of statements executed before the first execution of an image control statement, between the execution of two image control statements, or after the last execution of an image control statement is a segment. The segment executed immediately before the execution of an image control statement includes the evaluation of all expressions within the statement.

By execution of image control statements or user-defined ordering (11.6.5), the program can ensure that the execution of the \( i^{th} \) segment on image \( P_i \), either precedes or succeeds the execution of the \( j^{th} \) segment on another image \( Q_j \). If the program does not ensure this, segments \( P_i \) and \( Q_j \) are unordered; depending on the relative execution speeds of the images, some or all of the execution of the segment \( P_i \) may take place at the same time as some or all of the execution of the segment \( Q_j \).

A coarray may be referenced or defined by execution of an atomic subroutine during the execution of a segment that is unordered relative to the execution of a segment in which the coarray is referenced or defined by execution of an atomic subroutine. An event variable may be referenced or defined during the execution of a segment that is unordered relative to the execution of another segment in which that event variable is defined. Otherwise,

- if a variable is defined or becomes undefined on an image in a segment, it shall not be referenced, defined, or become undefined in a segment on another image unless the segments are ordered,
- if the allocation of an allocatable subobject of a coarray or the pointer association of a pointer subobject of a coarray is changed on an image in a segment, that subobject shall not be referenced, defined, or have its allocation or association status, dynamic type, array bounds, shape, or a deferred type parameter value inquired about in a segment on another image unless the segments are ordered, and
• if a procedure invocation on image P is in execution in segments \( P_i, P_{i+1}, \ldots, P_k \) and defines a noncoarray dummy argument, the effective argument shall not be referenced, defined, or become undefined on another image Q in a segment \( Q_j \) unless \( Q_j \) precedes \( P_i \) or succeeds \( P_k \).

**NOTE 11.32**

The set of all segments on all images is partially ordered: the segment \( P_i \) precedes segment \( Q_j \) if and only if there is a sequence of segments starting with \( P_i \) and ending with \( Q_j \) such that each segment of the sequence precedes the next either because they are consecutive segments on the same image or because of the execution of image control statements.

**NOTE 11.33**

If the segments \( S_1, S_2, \ldots, S_k \) on the distinct images \( P_1, P_2, \ldots, P_k \) are all unordered with respect to each other, it is expected that the processor will ensure that each of these images is provided with an equitable share of resources for executing its segment.

**NOTE 11.34**

Because of the restrictions on references and definitions in unordered segments, the processor can apply code motion optimizations within a segment as if it were the only image in execution, provided calls to atomic subroutines are not involved.

**NOTE 11.35**

The model upon which the interpretation of a program is based is that there is a permanent memory location for each coarray and that all images on which it is established can access it.

In practice, apart from executions of atomic subroutines, the processor could make a copy of a nonvolatile coarray on an image (in cache or a register, for example) and, as an optimization, defer copying a changed value back to the permanent memory location while it is still being used. Since the variable is not volatile, it is safe to defer this transfer until the end of the segment and thereafter to reload from permanent memory any coarray that was not defined within the segment. It might not be safe to defer these actions beyond the end of the segment since another image might reference the variable then.

The value of the ATOM argument of an atomic subroutine might be accessed or modified by another concurrently executing image. Therefore, execution of an atomic subroutine that references the ATOM argument cannot rely on a local copy, but instead always gets its value from its permanent memory location. Execution of an atomic subroutine that defines the ATOM argument does not complete until the value of its ATOM argument has been sent to its permanent memory location.

**NOTE 11.36**

The incorrect sequencing of image control statements can suspend execution indefinitely. For example, one image might be executing a SYNC ALL statement while another is executing an ALLOCATE statement for a coarray.

### 11.6.3 SYNC ALL statement

\[ \text{sync-all-stmt} \quad \text{is} \quad \text{SYNC ALL [ ( [ \text{sync-stat-list} ] ) ]} \]

\[ \text{sync-stat} \quad \text{is} \quad \text{STAT} = \text{stat-variable} \]

or \[ \text{ERRMSG} = \text{errmsg-variable} \]

C1170 No specifier shall appear more than once in a given sync-stat-list.

C1171 A stat-variable or errmsg-variable in a sync-stat shall not be a coindexed object.

The STAT= and ERRMSG= specifiers for image control statements are described in 11.6.11.
Successful execution of a SYNC ALL statement performs a synchronization of all images in the current team. Execution on an image, M, of the segment following the SYNC ALL statement is delayed until each other image in the current team has executed a SYNC ALL statement as many times as has image M in this team. The segments that executed before the SYNC ALL statement on an image precede the segments that execute after the SYNC ALL statement on another image.

NOTE 11.37

The processor might have special hardware or employ an optimized algorithm to make the SYNC ALL statement execute efficiently.

Here is a simple example of its use. Image 1 reads data and broadcasts it to other images:

```fortran
REAL :: P[*]
...
SYNC ALL
IF (THIS_IMAGE()==1) THEN
  READ (*,*) P
  DO I = 2, NUM_IMAGES()
    P[I] = P
  END DO
END IF
SYNC ALL
```

11.6.4 SYNC IMAGES statement

R1166 sync-images-stmt is SYNC IMAGES (image-set [, sync-stat-list ] )
R1167 image-set is int-expr or *
C1172 An image-set that is an int-expr shall be scalar or of rank one.
C1173 The value of image-set shall not depend on the value of stat-variable or errmsg-variable.

If image-set is an array expression, the value of each element shall be positive and not greater than the number of images in the current team, and there shall be no repeated values.

If image-set is a scalar expression, its value shall be positive and not greater than the number of images in the current team.

An image-set that is an asterisk specifies all images in the current team.

Execution of a SYNC IMAGES statement performs a synchronization of the image with each of the other images in the image-set. Executions of SYNC IMAGES statements on images M and T correspond if the number of times image M has executed a SYNC IMAGES statement in the current team with T in its image set is the same as the number of times image T has executed a SYNC IMAGES statement with M in its image set in this team. The segments that executed before the SYNC IMAGES statement on either image precede the segments that execute after the corresponding SYNC IMAGES statement on the other image.

NOTE 11.38

A SYNC IMAGES statement that specifies the single image index value THIS_IMAGE ( ) in its image set is allowed. This simplifies writing programs for an arbitrary number of images by allowing correct execution in the limiting case of the number of images being equal to one.
NOTE 11.39

In a program that uses `SYNC ALL` as its only synchronization mechanism, every `SYNC ALL` statement could be replaced by a `SYNC IMAGES (*)` statement, but `SYNC ALL` might give better performance.

`SYNC IMAGES` statements are not required to specify the entire `image` set, or even the same `image` set, on all `images` participating in the synchronization. In the following example, `image` 1 will wait for each of the other `images` to complete its use of the data. The other `images` wait for `image` 1 to set up the data, but do not wait on any other `image`.

```
IF (THIS_IMAGE() == 1) then
  ! Set up coarray data needed by all other images.
  SYNC IMAGES(*)
ELSE
  SYNC IMAGES(1)
  ! Use the data set up by image 1.
END IF
```

When the following example runs on five or more `images`, each `image` synchronizes with both of its neighbors, in a circular fashion.

```
INTEGER :: up, down
...
IF (NUM_IMAGES () > 1) THEN
  up = THIS_IMAGE () + 1; IF (up>iNUM_IMAGES ()) up = 1
  down = THIS_IMAGE () - 1; IF (down==0) down = NUM_IMAGES ()
  SYNC IMAGES ( (/ up, down /) )
END IF
```

This might appear to have the same effect as `SYNC ALL` but there is no ordering between the preceding and succeeding segments on non-adjacent `images`. For example, the segment preceding the `SYNC IMAGES` statement on `image` 3 will be ordered before those succeeding it on `images` 2 and 4, but not those on `images` 1 and 5.

NOTE 11.40

In the following example, each `image` synchronizes with its neighbor.

```
INTEGER :: ME, NE, STEP, NSTEPS
NE = NUM_IMAGES()
ME = THIS_IMAGE()
  ! Initial calculation
SYNC ALL
DO STEP = 1, NSTEPS
  IF (ME > 1) SYNC IMAGES(ME-1)
    ! Perform calculation
  IF (ME < NE) SYNC IMAGES(ME+1)
END DO
SYNC ALL
```

The calculation starts on `image` 1 since all the others will be waiting on `SYNC IMAGES (ME−1)`. When this is done, `image` 2 can start and `image` 1 can perform its second calculation. This continues until they are all executing different steps at the same time. Eventually, `image` 1 will finish and then the others will finish one by one.
11.6.5 SYNC MEMORY statement

Execution of a SYNC MEMORY statement ends one segment and begins another; those two segments can be ordered by a user-defined way with respect to segments on other images.

R1168 `sync-memory-stmt` is `SYNC MEMORY [ ( [ sync-stat-list ] ) ]`

If, by execution of statements on image P,

- a variable X on image Q is defined, referenced, becomes undefined, or has its allocation status, pointer association status, array bounds, dynamic type, or type parameters changed or inquired about by execution of a statement,
- that statement precedes a successful execution of a SYNC MEMORY statement, and
- a variable Y on image Q is defined, referenced, becomes undefined, or has its allocation status, pointer association status, array bounds, dynamic type, or type parameters changed or inquired about by execution of a statement that succeeds execution of that SYNC MEMORY statement,

then the action regarding X on image Q precedes the action regarding Y on image Q.

User-defined ordering of segment $P_i$ on image P to precede segment $Q_j$ on image Q occurs when

- image P executes an image control statement that ends segment $P_i$, and then executes statements that initiate a cooperative synchronization between images P and Q, and
- image Q executes statements that complete the cooperative synchronization between images P and Q and then executes an image control statement that begins segment $Q_j$.

Execution of the cooperative synchronization between images P and Q shall include a dependency that forces execution on image P of the statements that initiate the synchronization to precede the execution on image Q of the statements that complete the synchronization. The mechanisms available for creating such a dependency are processor dependent.

NOTE 11.41
SYNC MEMORY usually suppresses compiler optimizations that might reorder memory operations across the segment boundary defined by the SYNC MEMORY statement and ensures that all memory operations initiated in the preceding segments in its image complete before any memory operations in the subsequent segment in its image are initiated. It needs to do this unless it can establish that failure to do so could not alter processing on another image.

NOTE 11.42
SYNC MEMORY can be used to implement specialized schemes for segment ordering. For example, the user might have access to an external procedure that performs synchronization between images. That library procedure might not be aware of the mechanisms used by the processor to manage remote data references and definitions, and therefore not, by itself, be able to ensure the correct memory state before and after its reference. The SYNC MEMORY statement provides the needed memory ordering that enables the safe use of the external synchronization routine. For example:

```fortran
INTEGER :: IAM
REAL :: X[*]

IAM = THIS_IMAGE ()
IF (IAM == 1) X = 1.0
SYNC MEMORY
CALL EXTERNAL_SYNC ()
SYNC MEMORY
IF (IAM == 2) WRITE (*,*) X[1]
```
NOTE 11.42 (cont.)

where executing the subroutine EXTERNAL_SYNC has an image synchronization effect similar to executing a SYNC ALL statement.

11.6.6 SYNCE TEAM statement

R1169  sync-team-stmt  is  SYNC TEAM ( team-variable [, sync-stat-list] )

1 The team-variable shall be defined with a value that represents an ancestor team, the current team, or a team whose parent is the current team.

2 Successful execution of a SYNCE TEAM statement performs a synchronization of the team specified by team-variable. Execution on an image, M, of the segment following the SYNCE TEAM statement is delayed until each other image of the specified team has executed a SYNCE TEAM statement specifying the same team as many times as has image M in this team. The segments that executed before the SYNCE TEAM statement on an image precede the segments that execute after the corresponding SYNCE TEAM statement on another image.

NOTE 11.43

A SYNCE TEAM statement synchronizes a particular team whereas a SYNCE ALL statement synchronizes the current team.

11.6.7 EVENT POST statement

R1170  event-post-stmt  is  EVENT POST ( event-variable [, sync-stat-list] )

1 The EVENT POST statement posts an event.

R1171  event-variable  is  scalar-variable

C1174 (R1171) An event-variable shall be of type EVENT_TYPE from the intrinsic module ISO_FORTRAN_ENV (16.10.2).

2 The event-variable shall not depend on the value of stat-variable or errmsg-variable.

3 Successful execution of an EVENT POST statement atomically increments the count of the event variable by one. If an error condition occurs during execution of an EVENT POST statement, the value of the count of the event variable is processor dependent. The completion of an EVENT POST statement does not depend on the execution of a corresponding EVENT WAIT statement.

11.6.8 EVENT WAIT statement

R1172  event-wait-stmt  is  EVENT WAIT ( event-variable [, event-wait-spec-list] )

R1173  event-wait-spec  is  until-spec or sync-stat

R1174 (R1172) The event-variable in an event-wait-stmt shall not be coindexed.

C1175 (R1172) No specifier shall appear more than once in an event-wait-spec-list.

C1176 (R1172) The event-variable shall not depend on the value of stat-variable or errmsg-variable.

2 The event-variable shall not depend on the value of stat-variable or errmsg-variable.

3 Execution of an EVENT WAIT statement consists of the following sequence of actions:
1. if the UNTIL_COUNT= specifier does not appear, the threshold value is set to one; otherwise, the threshold value is set to the maximum of the value of the scalar-int-expr and one;

2. the executing image waits until the count of the event variable is greater than or equal to the threshold value or an error condition occurs;

3. if no error condition occurs, the count of the event variable is atomically decremented by the threshold value.

4. If an error occurs during execution of an EVENT WAIT statement, the value of the count of its event variable is processor dependent.

5. An EVENT POST statement execution is initially unsatisfied. Successful execution of an EVENT WAIT statement with a threshold of \( k \) satisfies the first \( k \) unsatisfied EVENT POST statement executions for that event variable. This EVENT WAIT statement execution causes the segment following the EVENT WAIT statement execution to succeed the segments preceding those \( k \) EVENT POST statement executions.

11.6.9 FORM TEAM statement

1. The FORM TEAM statement creates new teams whose parent team is the current team.

R1175 form-team-stmt is FORM TEAM ( team-number, team-variable □ □ □ □ ] form-team-spec-list ] )

R1176 team-number is scalar-int-expr

R1177 form-team-spec is NEW_INDEX = scalar-int-expr or sync-stat

C1177 No specifier shall appear more than once in given a form-team-spec-list.

2. Successful execution of a FORM TEAM statement creates a new team for each unique team-number value specified by the active images of the current team. The value of team-number shall be positive. Each executing image will belong to the team whose team number is equal to the value of team-number on that image, and the team-variable becomes defined with a value that specifies that team.

3. The value of the scalar-int-expr in a NEW_INDEX= specifier specifies the image index that the executing image will have in its new team. It shall be positive, less than or equal to the number of images in the team, and different from the value specified by every other image that belongs to that team.

4. If the NEW_INDEX= specifier does not appear, the image index of the executing image in the new team is processor dependent. This image index will be positive, less than or equal to the number of images in the team, and different from that of every other image in the team.

5. If the FORM TEAM statement is executed on one image, the same statement shall be executed on all active images of the current team. When a FORM TEAM statement is executed, there is an implicit synchronization of all active images in the current team. On those images, execution of the segment following the statement is delayed until all other active images in the current team have executed the same statement the same number of times in this team. If an error condition other than detection of a failed image occurs, the team variable becomes undefined.

6. If execution of a FORM TEAM statement assigns the value STAT_FAILED_IMAGE to the stat-variable, the effect is the same as for the successful execution of FORM TEAM except for the value assigned to stat-variable.

NOTE 11.44

<table>
<thead>
<tr>
<th>Executing the statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORM TEAM ( 2 - MOD (THIS_IMAGE (), 2), ODD_EVEN )</td>
</tr>
</tbody>
</table>

214 ISO/IEC JTC 1/SC 22/WG5/N2118
NOTE 11.44 (cont.)

will create two subteams of the current team, with images whose image index is odd being in the team with number 1, and those with an even image index being in the team with number 2.

NOTE 11.45

If the current team consists of $P^2$ images, with corresponding coarrays on each image representing parts of a larger array spread over a $P \times P$ square, the following code will establish teams for the rows with image indices equal to the column indices.

USE, INTRINSIC :: ISO_FORTRAN_ENV
TYPE(TEAM_TYPE) :: ROW
REAL :: A [P, *]
INTEGER :: ME (2)
ME (: ) = THIS_IMAGE (A)
FORM TEAM (ME(1), ROW, NEW_INDEX=ME(2))

11.6.10 LOCK and UNLOCK statements

R1178 lock-stmt is LOCK ( lock-variable [ , lock-stat-list ] )
R1179 lock-stat is ACQUIRED_LOCK = scalar-logical-variable
or sync-stat

C1178 No specifier shall appear more than once in a given lock-stat-list.

R1180 unlock-stmt is UNLOCK ( lock-variable [ , sync-stat-list ] )
R1181 lock-variable is scalar-variable

C1179 (R1181) A lock-variable shall be of type LOCK_TYPE from the intrinsic module ISO_FORTRAN_ENV (16.10.2.19).

1 The lock-variable shall not depend on the value of stat-variable, errmsg-variable, or the scalar-logical-variable in the ACQUIRED_LOCK= specifier. The scalar-logical-variable shall not depend on the value of the lock-variable, stat-variable, or errmsg-variable.

2 A lock variable is unlocked if and only if the value of each component is the same as its default value. If it has any other value, it is locked. A lock variable is locked by an image if it was locked by execution of a LOCK statement on that image, has not been subsequently unlocked by execution of an UNLOCK statement on the same image, and that image has not failed.

3 Successful execution of a LOCK statement without an ACQUIRED_LOCK= specifier causes the lock variable to become locked by that image. If the lock variable is already locked by another image, that LOCK statement causes the lock variable to become locked after the other image causes the lock variable to become unlocked.

4 If the lock variable is unlocked, successful execution of a LOCK statement with an ACQUIRED_LOCK= specifier causes the lock variable to become locked by that image and the scalar logical variable to become defined with the value true. If the lock variable is already locked by a different image, successful execution of a LOCK statement with an ACQUIRED_LOCK= specifier leaves the lock variable unchanged and causes the scalar logical variable to become defined with the value false.

5 Successful execution of an UNLOCK statement causes the lock variable to become unlocked. Failure of an image causes all lock variables that are locked by that image to become unlocked.

6 During execution of the program, the value of a lock variable changes through a sequence of locked and unlocked states due to the execution of LOCK and UNLOCK statements, and by failure of an image while it is locked by
that image. If a lock variable becomes unlocked by execution of an UNLOCK statement on image M and next becomes locked by execution of a LOCK statement on image T, the segments preceding the UNLOCK statement on image M precede the segments following the LOCK statement on image T. Execution of a LOCK statement that does not cause the lock variable to become locked does not affect segment ordering.

An error condition occurs if the lock variable in a LOCK statement is already locked by the executing image. An error condition occurs if the lock variable in an UNLOCK statement is not already locked by the executing image. If an error condition occurs during execution of a LOCK or UNLOCK statement, the value of the lock variable is not changed and the value of the ACQUIRED_LOCK variable, if any, is not changed.

**NOTE 11.46**
A lock variable is effectively defined atomically by a LOCK or UNLOCK statement. If LOCK statements on two images both attempt to acquire a lock, one will succeed and the other will either fail if an ACQUIRED_LOCK= specifier appears, or will wait until the lock is later released if an ACQUIRED_LOCK= specifier does not appear.

**NOTE 11.47**
An image might wait for a LOCK statement to successfully complete for a long period of time if other images frequently lock and unlock the same lock variable. This situation might result from executing LOCK statements with ACQUIRED_LOCK= specifiers inside a spin loop.

**NOTE 11.48**
The following example illustrates the use of LOCK and UNLOCK statements to manage a work queue:

```fortran
USE, INTRINSIC :: ISO_FORTRAN_ENV

TYPE(LOCK_TYPE) :: queue_lock[*] ! Lock on each image to manage its work queue
INTEGER :: work_queue_size[*]

TYPE(Task) :: work_queue(100)[*] ! List of tasks to perform

TYPE(Task) :: job ! Current task working on
INTEGER :: me

me = THIS_IMAGE()

DO
  ! Process the next item in your work queue

  LOCK (queue_lock) ! New segment A starts
  ! This segment A is ordered with respect to
  ! segment B executed by image me-1 below because of lock exclusion
  IF (work_queue_size>0) THEN
    ! Fetch the next job from the queue
    job = work_queue(work_queue_size)
    work_queue_size = work_queue_size-1
  END IF

  UNLOCK (queue_lock) ! Segment ends

  ! Add a new task on neighbors queue:
  LOCK(queue_lock[me+1]) ! Starts segment B
  ! This segment B is ordered with respect to
  ! segment A executed by image me+1 above because of lock exclusion
  IF (work_queue_size[me+1]<SIZE (work_queue)) THEN
    work_queue_size[me+1] = work_queue_size[me+1]+1
  END IF

  ! Actually process the task
  ...

  ! Add a new task on neighbors queue:
  LOCK(queue_lock[me+1]) ! Starts segment B
  ! This segment B is ordered with respect to
  ! segment A executed by image me+1 above because of lock exclusion
  IF (work_queue_size[me+1]<SIZE (work_queue)) THEN
    work_queue_size[me+1] = work_queue_size[me+1]+1
  END IF
```

ISO/IEC JTC 1/SC 22/WG5/N2118
NOTE 11.48 (cont.)

\[
\text{work\_queue(work\_queue\_size[me+1])}[me+1] = \text{job}
\]

END IF

UNLOCK (queue\_lock[me+1]) ! Ends segment B

END DO

11.6.11 STAT= and ERRMSG= specifiers in image control statements

1 In an image control statement, the \textit{stat-variable} in a \textit{sync-stat} shall not depend on the value of an \textit{errmsg-variable} in a \textit{sync-stat}, \textit{event-variable}, \textit{lock-variable}, \textit{team-variable}, or the \textit{scalar-logical-variable} in the ACQUIRED\_LOCK= specifier. The \textit{errmsg-variable} in a \textit{sync-stat} shall not depend on the value of a \textit{stat-variable} in a \textit{sync-stat}, \textit{event-variable}, \textit{lock-variable}, \textit{team-variable}, or the \textit{scalar-logical-variable} in the ACQUIRED\_LOCK= specifier.

2 If a STAT= specifier appears in a \textit{sync-stat} in an image control statement, the \textit{stat-variable} is assigned the value zero if execution of the statement is successful.

3 If the STAT= specifier appears in a \textit{sync-stat} in an EVENT \textit{WAIT} or \textit{SYNC MEMORY} statement and an error condition occurs, \textit{stat-variable} is assigned a processor-dependent positive value that is different from the value of STAT\_FAILED\_IMAGE (16.10.2.26) and STAT\_STOPPED\_IMAGE (16.10.2.29) from the intrinsic module ISO\_FORTRAN\_ENV (16.10.2).

4 The images involved in execution of an END \textit{TEAM}, FORM \textit{TEAM}, or \textit{SYNC ALL} statement are those in the current team. The images involved in execution of a \textit{CHANGE TEAM} or \textit{SYNC TEAM} statement are those of the specified team. The images involved in execution of a \textit{SYNC IMAGES} statement are the images specified and the executing image. The images involved in execution of an EVENT \textit{POST} statement are the image on which the event variable is located and the executing image.

5 If the STAT= specifier appears in a \textit{sync-stat} in a \textit{CHANGE TEAM}, END \textit{TEAM}, EVENT \textit{POST}, FORM \textit{TEAM}, \textit{SYNC ALL}, \textit{SYNC IMAGES}, or \textit{SYNC TEAM} statement,

- if one of the images involved has stopped, \textit{stat-variable} is assigned the value STAT\_STOPPED\_IMAGE (16.10.2.29) from the intrinsic module ISO\_FORTRAN\_ENV;
- otherwise, if one of the images involved has failed and no other error condition occurs, the intended action is performed on the active images involved and \textit{stat-variable} is assigned the value STAT\_FAILED\_IMAGE (16.10.2.26) from the intrinsic module ISO\_FORTRAN\_ENV;
- otherwise, if any other error condition occurs, \textit{stat-variable} is assigned a processor-dependent positive value that is different from the values of STAT\_STOPPED\_IMAGE and STAT\_FAILED\_IMAGE.

6 If the STAT= specifier appears in a \textit{sync-stat} in a \textit{SYNC ALL}, \textit{SYNC IMAGES}, or \textit{SYNC TEAM} statement and the error condition STAT\_STOPPED\_IMAGE occurs, the effect is the same as that of executing the SYNC MEMORY statement, except for defining the \textit{stat-variable}.

7 If the STAT= specifier appears in a \textit{sync-stat} in a \textit{LOCK} statement,

- if the image on which the lock variable is located has failed, the \textit{stat-variable} becomes defined with the value STAT\_FAILED\_IMAGE;
- otherwise, if the lock variable is locked by the executing image, the \textit{stat-variable} becomes defined with the value of STAT\_LOCKED (16.10.2.27) from the intrinsic module ISO\_FORTRAN\_ENV;
- otherwise, if the lock variable is unlocked because of the failure of the image that locked it, \textit{stat-variable} becomes defined with the value STAT\_UNLOCKED\_FAILED\_IMAGE (16.10.2.31) from the intrinsic module ISO\_FORTRAN\_ENV.

8 If the STAT= specifier appears in a \textit{sync-stat} in an UNLOCK statement,
• if the image on which the lock variable is located has failed, the stat-variable becomes defined with the value STAT_FAILED_IMAGE;
• otherwise, if the lock variable has the value unlocked, the stat-variable becomes defined with the value of STAT_UNLOCKED (16.10.2.30) from the intrinsic module ISO_FORTRAN_ENV;
• otherwise, if the lock variable is locked by a different image, the stat-variable becomes defined with the value STAT_LOCKED_OTHER_IMAGE (16.10.2.28) from the intrinsic module ISO_FORTRAN_ENV.

If any other error condition occurs during execution of a LOCK or UNLOCK statement, the specified variable becomes defined with a positive integer value that is different from STAT_LOCKED, STAT_LOCKED_OTHER_IMAGE, STAT_UNLOCKED, and STAT_UNLOCKED_FAILED_IMAGE.

If an image completes execution of a CRITICAL statement that has a sync-stat that is a STAT= specifier and the previous image to have entered the construct failed while executing it, the stat-variable becomes defined with the value STAT_FAILED_IMAGE and execution of the construct continues normally. If any other error condition occurs during execution of a CRITICAL statement that has a STAT= specifier, the stat-variable becomes defined with a processor-dependent value other than STAT_FAILED_IMAGE.

If an error condition occurs during execution of an image control statement that does not contain the STAT= specifier in a sync-stat, error termination is initiated.

If an ERRMSG= specifier appears in an image control statement and an error condition occurs, errmsg-variable is assigned an explanatory message, as if by intrinsic assignment. If no such condition occurs, the definition status and value of errmsg-variable are unchanged.

The set of error conditions that can occur in an image control statement is processor dependent.

NOTE 11.49

A processor might detect communication failure between images and treat it as an error condition. A processor might also treat an invalid set of images in a SYNC IMAGES statement as an error condition.
12 Input/output statements

12.1 Input/output concepts

Input statements provide the means of transferring data from external media to internal storage or from an internal file to internal storage. This process is called reading. Output statements provide the means of transferring data from internal storage to external media or from internal storage to an internal file. This process is called writing. Some input/output statements specify that editing of the data is to be performed.

In addition to the statements that transfer data, there are auxiliary input/output statements to manipulate the external medium, or to describe or inquire about the properties of the connection to the external medium.

The input/output statements are the BACKSPACE, CLOSE, ENDFILE, FLUSH, INQUIRE, OPEN, PRINT, READ, REWIND, WAIT, and WRITE statements.

A file is composed of either a sequence of file storage units (12.3.5) or a sequence of records, which provide an extra level of organization to the file. A file composed of records is called a record file. A file composed of file storage units is called a stream file. A processor may allow a file to be viewed both as a record file and as a stream file; in this case the relationship between the file storage units when viewed as a stream file and the records when viewed as a record file is processor dependent.

A file is either an external file (12.3) or an internal file (12.4).

12.2 Records

12.2.1 Definition of a record

A record is a sequence of values or a sequence of characters. For example, a line on a terminal is usually considered to be a record. However, a record does not necessarily correspond to a physical entity. There are three kinds of records:

(1) formatted;
(2) unformatted;
(3) endfile.

NOTE 12.1
What is called a “record” in Fortran is commonly called a “logical record”. There is no concept in Fortran of a “physical record.”

12.2.2 Formatted record

A formatted record consists of a sequence of characters that are representable in the processor; however, a processor may prohibit some control characters (6.1.1) from appearing in a formatted record. The length of a formatted record is measured in characters and depends primarily on the number of characters put into the record when it is written; however, it may depend on the processor and the external medium. The length may be zero. Formatted records shall be read or written only by formatted input/output statements.

12.2.3 Unformatted record

An unformatted record consists of a sequence of values in a processor-dependent form and may contain data of any type or may contain no data. The length of an unformatted record is measured in file storage units.
1 (12.3.5) and depends on the output list (12.6.3) used when it is written, as well as on the processor and the
external medium. The length may be zero. Unformatted records may be read or written only by unformatted
input/output statements.

12.2.4 Endfile record

1 An endfile record is written explicitly by the ENDFILE statement; the file shall be connected for sequential
access. An endfile record is written implicitly to a file connected for sequential access when the most recent data
transfer statement referring to the file is an output statement, no intervening file positioning statement referring
to the file has been executed, and

• a REWIND or BACKSPACE statement references the unit to which the file is connected, or
• the unit is closed, either explicitly by a CLOSE statement, implicitly by normal termination, or implicitly
by another OPEN statement for the same unit.

2 An endfile record may occur only as the last record of a file. An endfile record does not have a length property.

NOTE 12.2
An endfile record does not necessarily have any physical embodiment. The processor can use a record count
or any other means to register the position of the file at the time an ENDFILE statement is executed, so
that it can take appropriate action when that position is reached again during a read operation. The endfile
record, however it is implemented, is considered to exist for the BACKSPACE statement (12.8.2).

12.3 External files

12.3.1 External file concepts

1 An external file is any file that exists in a medium external to the program.

2 At any given time, there is a processor-dependent set of allowed access methods, a processor-dependent set of
allowed forms, a processor-dependent set of allowed actions, and a processor-dependent set of allowed record
lengths for a file.

NOTE 12.3
For example, the processor-dependent set of allowed actions for a printer would likely include the write
action, but not the read action.

3 A file may have a name; a file that has a name is called a named file. The name of a named file is represented by
a character string value. The set of allowable names for a file is processor dependent. Whether a named file on
one image is the same as a file with the same name on another image is processor dependent.

NOTE 12.4
If different files are needed on each image, using a different file name on each image will improve portability
of the code. One technique is to incorporate the image index as part of the name.

4 An external file that is connected to a unit has a position property (12.3.4).

NOTE 12.5
For more explanatory information on external files, see C.7.1.

12.3.2 File existence

1 At any given time, there is a processor-dependent set of external files that exist for a program. A file may be
known to the processor, yet not exist for a program at a particular time.
To create a file means to cause a file to exist that did not exist previously. To delete a file means to terminate the existence of the file.

All input/output statements may refer to files that exist. A CLOSE, ENDFILE, FLUSH, INQUIRE, OPEN, PRINT, REWIND, or WRITE statement is permitted to refer to a file that does not exist. No other input/output statement shall refer to a file that does not exist. Execution of a WRITE, PRINT, or ENDFILE statement referring to a preconnected file that does not exist creates the file. This file is a different file from one preconnected on any other image.

12.3.3 File access

12.3.3.1 File access methods

There are three methods of accessing the data of an external file: sequential, direct, and stream. Some files may have more than one allowed access method; other files may be restricted to one access method.

NOTE 12.6
For example, a processor might provide only sequential access to a file on magnetic tape. Thus, the set of allowed access methods depends on the file and the processor.

The method of accessing a file is determined when the file is connected to a unit (12.5.4) or when the file is created if the file is preconnected (12.5.5).

12.3.3.2 Sequential access

Sequential access is a method of accessing the records of an external record file in order.

While connected for sequential access, an external file has the following properties.

- The order of the records is the order in which they were written if the direct access method is not a member of the set of allowed access methods for the file. If the direct access method is also a member of the set of allowed access methods for the file, the order of the records is the same as that specified for direct access. In this case, the first record accessible by sequential access is the record whose record number is 1 for direct access. The second record accessible by sequential access is the record whose record number is 2 for direct access, etc. A record that has not been written since the file was created shall not be read.
- The records of the file are either all formatted or all unformatted, except that the last record of the file may be an endfile record. Unless the previous reference to the file was an output statement, the last record, if any, of the file shall be an endfile record.
- The records of the file shall be read or written only by sequential access data transfer statements.

12.3.3.3 Direct access

Direct access is a method of accessing the records of an external record file in arbitrary order.

While connected for direct access, an external file has the following properties.

- Each record of the file is uniquely identified by a positive integer called the record number. The record number of a record is specified when the record is written. Once established, the record number of a record cannot be changed. The order of the records is the order of their record numbers.
- The records of the file are either all formatted or all unformatted. If the sequential access method is also a member of the set of allowed access methods for the file, its endfile record, if any, is not considered to be part of the file while it is connected for direct access. If the sequential access method is not a member of the set of allowed access methods for the file, the file shall not contain an endfile record.
- The records of the file shall be read or written only by direct access data transfer statements.
- All records of the file have the same length.
Records need not be read or written in the order of their record numbers. Any record may be written into the file while it is connected to a unit. For example, it is permissible to write record 3, even though records 1 and 2 have not been written. Any record may be read from the file while it is connected to a unit, provided that the record has been written since the file was created, and if a READ statement for this connection is permitted.

The records of the file shall not be read or written using list-directed formatting (13.10), namelist formatting (13.11), or a nonadvancing data transfer statement (12.3.4.2).

NOTE 12.7
A record cannot be deleted; however, a record can be rewritten.

12.3.3.4 Stream access

Stream access is a method of accessing the file storage units (12.3.5) of an external stream file.

The properties of an external file connected for stream access depend on whether the connection is for unformatted or formatted access. While connected for stream access, the file storage units of the file shall be read or written only by stream access data transfer statements.

While connected for unformatted stream access, an external file has the following properties.

- Each file storage unit in the file is uniquely identified by a positive integer called the position. The first file storage unit in the file is at position 1. The position of each subsequent file storage unit is one greater than that of its preceding file storage unit.
- If it is possible to position the file, the file storage units need not be read or written in order of their position. For example, it might be permissible to write the file storage unit at position 3, even though the file storage units at positions 1 and 2 have not been written. Any file storage unit may be read from the file while it is connected to a unit, provided that the file storage unit has been written since the file was created, and if a READ statement for this connection is permitted.

While connected for formatted stream access, an external file has the following properties.

- Some file storage units of the file may contain record markers; this imposes a record structure on the file in addition to its stream structure. There might or might not be a record marker at the end of the file. If there is no record marker at the end of the file, the final record is incomplete.
- No maximum length (12.5.6.15) is applicable to these records.
- Writing an empty record with no record marker has no effect.
- Each file storage unit in the file is uniquely identified by a positive integer called the position. The first file storage unit in the file is at position 1. The relationship between positions of successive file storage units is processor dependent; not all positive integers need correspond to valid positions.
- If it is possible to position the file, the file position can be set to a position that was previously identified by the POS= specifier in an INQUIRE statement.
- A processor may prohibit some control characters (6.1.1) from appearing in a formatted stream file.

NOTE 12.8
Because the record structure is determined from the record markers that are stored in the file itself, an incomplete record at the end of the file is necessarily not empty.

NOTE 12.9
There might be some character positions in the file that do not correspond to characters written; this is because on some processors a record marker could be written to the file as a carriage-return/line-feed or other sequence. The means of determining the position in a file connected for stream access is via the POS= specifier in an INQUIRE statement (12.10.2.22).
12.3.4 File position

12.3.4.1 General

1 Execution of certain input/output statements affects the position of an external file. Certain circumstances can
cause the position of a file to become indeterminate.

2 The initial point of a file is the position just before the first record or file storage unit. The terminal point is the
position just after the last record or file storage unit. If there are no records or file storage units in the file, the
initial point and the terminal point are the same position.

3 If a record file is positioned within a record, that record is the current record; otherwise, there is no current
record.

4 Let \( n \) be the number of records in the file. If \( 1 < i \leq n \) and a file is positioned within the \( i \)th record or between
the \( (i-1) \)th record and the \( i \)th record, the \( (i-1) \)th record is the preceding record. If \( n \geq 1 \) and the file is
positioned at its terminal point, the preceding record is the \( n \)th and last record. If \( n = 0 \) or if a file is positioned
at its initial point or within the first record, there is no preceding record.

5 If \( 1 \leq i < n \) and a file is positioned within the \( i \)th record or between the \( i \)th and \( (i+1) \)th record, the \( (i+1) \)th
record is the next record. If \( n \geq 1 \) and the file is positioned at its initial point, the first record is the next record.
If \( n = 0 \) or if a file is positioned at its terminal point or within the \( n \)th (last) record, there is no next record.

6 For a file connected for stream access, the file position is either between two file storage units, at the initial point
of the file, at the terminal point of the file, or undefined.

12.3.4.2 Advancing and nonadvancing input/output

1 An advancing input/output statement always positions a record file after the last record read or written, unless
there is an error condition.

2 A nonadvancing input/output statement may position a record file at a character position within the current
record, or a subsequent record (13.8.2). Using nonadvancing input/output, it is possible to read or write a record
of the file by a sequence of data transfer statements, each accessing a portion of the record. If a nonadvancing
output statement leaves a file positioned within a current record and no further output statement is executed for
the file before it is closed or a BACKSPACE, ENDFILE, or REWIND statement is executed for it, the effect is
as if the output statement were the corresponding advancing output statement.

12.3.4.3 File position prior to data transfer

1 The positioning of the file prior to data transfer depends on the method of access: sequential, direct, or stream.

2 For sequential access on input, if there is a current record, the file position is not changed. Otherwise, the file is
positioned at the beginning of the next record and this record becomes the current record. Input shall not occur
if there is no next record or if there is a current record and the last data transfer statement accessing the file
performed output.

3 If the file contains an endfile record, the file shall not be positioned after the endfile record prior to data transfer.
However, a REWIND or BACKSPACE statement may be used to reposition the file.

4 For sequential access on output, if there is a current record, the file position is not changed and the current record
becomes the last record of the file. Otherwise, a new record is created as the next record of the file; this new
record becomes the last and current record of the file and the file is positioned at the beginning of this record.

5 For direct access, the file is positioned at the beginning of the record specified by the REC= specifier. This record
becomes the current record.

6 For stream access, the file is positioned immediately before the file storage unit specified by the POS= specifier;
if there is no POS= specifier, the file position is not changed.
File positioning for child data transfer statements is described in 12.6.4.8.

12.3.4.4 File position after data transfer

1 If an error condition (12.11) occurred, the position of the file is indeterminate. If no error condition occurred, but an end-of-file condition (12.11) occurred as a result of reading an endfile record, the file is positioned after the endfile record.

2 For unformatted stream input/output, if no error condition occurred, the file position is not changed. For unformatted stream output, if the file position exceeds the previous terminal point of the file, the terminal point is set to the file position.

NOTE 12.10
An unformatted stream output statement with a POS= specifier and an empty output list can have the effect of extending the terminal point of a file without actually writing any data.

3 For formatted stream input, if an end-of-file condition occurred, the file position is not changed.

4 For nonadvancing input, if no error condition or end-of-file condition occurred, but an end-of-record condition (12.11) occurred, the file is positioned after the record just read. If no error condition, end-of-file condition, or end-of-record condition occurred in a nonadvancing input statement, the file position is not changed. If no error condition occurred in a nonadvancing output statement, the file position is not changed.

5 In all other cases, the file is positioned after the record just read or written and that record becomes the preceding record.

6 For a formatted stream output statement, if no error condition occurred, the terminal point of the file is set to the next position after the highest-numbered position to which a datum was transferred by the statement.

NOTE 12.11
The highest-numbered position might not be the current one if the output involved a T, TL, TR, or X edit descriptor (13.8.1) and the statement is a nonadvancing output statement.

12.3.5 File storage units

1 A file storage unit is the basic unit of storage in a stream file or an unformatted record file. It is the unit of file position for stream access, the unit of record length for unformatted files, and the unit of file size for all external files.

2 Every value in a stream file or an unformatted record file shall occupy an integer number of file storage units; if the stream or record file is unformatted, this number shall be the same for all scalar values of the same type and type parameters. The number of file storage units required for an item of a given type and type parameters may be determined using the IOLength= specifier of the INQUIRE statement (12.10.3).

3 For a file connected for unformatted stream access, the processor shall not have alignment restrictions that prevent a value of any type from being stored at any positive integer file position.

4 The number of bits in a file storage unit is given by the constant FILE_STORAGE_SIZE (16.10.2.11) defined in the intrinsic module ISO_FORTRAN_ENV. It is recommended that the file storage unit be an 8-bit octet where this choice is practical.

NOTE 12.12
The requirement that every data value occupy an integer number of file storage units implies that data items inherently smaller than a file storage unit will require padding. This suggests that the file storage unit be small to avoid wasted space. Ideally, the file storage unit would be chosen such that padding is never required. A file storage unit of one bit would always meet this goal, but would likely be impractical because of the alignment requirements.
NOTE 12.12 (cont.)

The prohibition on alignment restrictions prohibits the processor from requiring data alignments larger than the file storage unit.

The 8-bit octet is recommended as a good compromise that is small enough to accommodate the requirements of many applications, yet not so small that the data alignment requirements are likely to cause significant performance problems.

12.4 Internal files

Internal files provide a means of transferring and converting data from internal storage to internal storage.

An internal file is a record file with the following properties.

- The file is a variable of default, ASCII, or ISO 10646 character kind that is not an array section with a vector subscript.
- A record of an internal file is a scalar character variable.
- If the file is a scalar character variable, it consists of a single record whose length is the same as the length of the scalar character variable. If the file is a character array, it is treated as a sequence of character array elements. Each array element, if any, is a record of the file. The ordering of the records of the file is the same as the ordering of the array elements in the array (9.5.3.2) or the array section (9.5.3.3). Every record of the file has the same length, which is the length of an element array in the element.
- A record of the internal file becomes defined by writing the record. If the number of characters written in a record is less than the length of the record, the remaining portion of the record is filled with blanks. The number of characters to be written shall not exceed the length of the record.
- A record may be read only if the record is defined.
- A record of an internal file may become defined (or undefined) by means other than an output statement. For example, the character variable may become defined by a character assignment statement.
- An internal file is always positioned at the beginning of the first record prior to data transfer, except for child data transfer statements (12.6.4.8). This record becomes the current record.
- The initial value of a connection mode (12.5.2) is the value that would be implied by an initial OPEN statement without the corresponding keyword.
- Reading and writing records shall be accomplished only by sequential access formatted data transfer statements.
- An internal file shall not be specified as the unit in a CLOSE, INQUIRE, or OPEN statement.

12.5 File connection

12.5.1 Referring to a file

A unit, specified by an io-unit, provides a means for referring to a file.

R1201 io-unit is file-unit-number
or *
or internal-file-variable

R1202 file-unit-number is scalar-int-expr

R1203 internal-file-variable is char-variable

C1201 (R1203) The char-variable shall not be an array section with a vector subscript.

C1202 (R1203) The char-variable shall be default character, ASCII character, or ISO 10646 character.
A unit is either an external unit or an internal unit. An external unit is used to refer to an external file and is specified by an asterisk or a file-unit-number. The value of file-unit-number shall be nonnegative, equal to one of the named constants INPUT_UNIT, OUTPUT_UNIT, or ERROR_UNIT of the intrinsic module ISO_FORTRAN_ENV (16.10.2), the unit argument of an active defined input/output procedure (12.6.4.8), or a NEWUNIT value (12.5.6.12). An internal unit is used to refer to an internal file and is specified by an internal-file-variable or a file-unit-number whose value is equal to the unit argument of an active defined input/output procedure. The value of a file-unit-number shall identify a valid unit.

On an image, the external unit identified by a particular value of a scalar-int-expr is the same external unit in all program units.

NOTE 12.13
In the example:

```fortran
SUBROUTINE A
   READ (6) X
...
SUBROUTINE B
   N = 6
   REWIND N
```

the value 6 used in both program units identifies the same external unit.

In a READ statement, an io-unit that is an asterisk identifies an external unit that is preconnected for sequential formatted input on image 1 in the initial team only (12.6.4.3). This unit is also identified by the value of the named constant INPUT_UNIT of the intrinsic module ISO_FORTRAN_ENV (16.10.2.13). In a WRITE statement, an io-unit that is an asterisk identifies an external unit that is preconnected for sequential formatted output. This unit is also identified by the value of the named constant OUTPUT_UNIT of the intrinsic module ISO_FORTRAN_ENV (16.10.2.22).

This document identifies a processor-dependent external unit for the purpose of error reporting. This unit shall be preconnected for sequential formatted output. The processor may define this to be the same as the output unit identified by an asterisk. This unit is also identified by a unit number defined by the named constant ERROR_UNIT of the intrinsic module ISO_FORTRAN_ENV.

NOTE 12.14
Even though OUTPUT_UNIT is connected to a separate file on each image, it is expected that the processor could merge the sequences of records from these files into a single sequence of records that is sent to the physical device associated with this unit, such as the user’s terminal. If ERROR_UNIT is associated with the same physical device, the sequences of records from files connected to ERROR_UNIT on each of the images could be merged into the same sequence generated from the OUTPUT_UNIT files. Otherwise, it is expected that the sequence of records in the files connected to ERROR_UNIT on each image could be merged into a single sequence of records that is sent to the physical device associated with ERROR_UNIT.

12.5.2 Connection modes

A connection for formatted input/output has several changeable modes: these are the blank interpretation mode (13.8.6), delimiter mode (13.10.4, 13.11.4.2), sign mode (13.8.4), decimal edit mode (13.8.8), input/output rounding mode (13.7.2.3.8), pad mode (12.6.4.5.3), and scale factor (13.8.5). A connection for unformatted input/output has no changeable modes.

Values for the modes of a connection are established when the connection is initiated. If the connection is initiated by an OPEN statement, the values are as specified, either explicitly or implicitly, by the OPEN statement. If the connection is initiated other than by an OPEN statement (that is, if the file is an internal file or preconnected file) the values established are those that would be implied by an initial OPEN statement without the corresponding keywords.
The scale factor cannot be explicitly specified in an OPEN statement; it is implicitly 0.

The modes of a connection to an external file may be changed by a subsequent OPEN statement that modifies the connection.

The modes of a connection may be temporarily changed by a corresponding keyword specifier in a data transfer statement or by an edit descriptor. Keyword specifiers take effect at the beginning of execution of the data transfer statement. Edit descriptors take effect when they are encountered in format processing. When a data transfer statement terminates, the values for the modes are reset to the values in effect immediately before the data transfer statement was executed.

12.5.3 Unit existence

At any given time, there is a processor-dependent set of external units that exist for an image.

All input/output statements are permitted to refer to units that exist. The CLOSE, INQUIRE, and WAIT statements are also permitted to refer to units that do not exist. No other input/output statement shall refer to a unit that does not exist.

12.5.4 Connection of a file to a unit

An external unit has a property of being connected or not connected. If connected, it refers to an external file. An external unit may become connected by preconnection or by the execution of an OPEN statement. The property of connection is symmetric; the unit is connected to a file if and only if the file is connected to the unit.

Every input/output statement except an OPEN, CLOSE, INQUIRE, or WAIT statement shall refer to a unit that is connected to a file and thereby make use of or affect that file.

A file may be connected and not exist (12.3.2).

NOTE 12.15

An example is a preconnected external file that has not yet been written.

A unit shall not be connected to more than one file at the same time. However, means are provided to change the status of an external unit and to connect a unit to a different file. It is processor dependent whether a file can be connected to more than one unit at the same time.

This document defines means of portable interoperation with C. C streams are described in 7.21.2 of ISO/IEC 9899:2011. Whether a unit can be connected to a file that is also connected to a C stream is processor dependent. If a unit is connected to a file that is also connected to a C stream, the results of performing input/output operations on such a file are processor dependent. It is processor dependent whether the files connected to the units INPUT_UNIT, OUTPUT_UNIT, and ERROR_UNIT correspond to the predefined C text streams standard input, standard output, and standard error. If a main program or procedure defined by means of Fortran and a main program or procedure defined by means other than Fortran perform input/output operations on the same external file, the results are processor dependent. A main program or procedure defined by means of Fortran and a main program or procedure defined by means other than Fortran can perform input/output operations on different external files without interference.

If input/output operations are performed on more than one unit while they are connected to the same external file, the results are processor dependent.

After an external unit has been disconnected by the execution of a CLOSE statement, it may be connected again within the same program to the same file or to a different file. After an external file has been disconnected by the execution of a CLOSE statement, it may be connected again within the same program to the same unit or to a different unit.
NOTE 12.16
The only means of referencing a file that has been disconnected is by the appearance of its name in an OPEN or INQUIRE statement. There might be no means of reconnecting an unnamed file once it is disconnected.

NOTE 12.17
For more explanatory information on file connection properties, see C.7.4.

12.5.5 Preconnection

1 Preconnection means that the unit is connected to a file at the beginning of execution of the program and therefore it may be specified in input/output statements without the prior execution of an OPEN statement.

12.5.6 OPEN statement

12.5.6.1 General

1 An OPEN statement initiates or modifies the connection between an external file and a specified unit. The OPEN statement may be used to connect an existing file to a unit, create a file that is preconnected, create a file and connect it to a unit, or change certain modes of a connection between a file and a unit.

2 An external unit may be connected by an OPEN statement in the main program or any subprogram.

3 If the file to be connected to the unit does not exist but is the same as the file to which the unit is preconnected, the modes specified by an OPEN statement become a part of the connection.

4 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 an OPEN statement.

5 If a unit is connected to a file that exists, execution of an OPEN statement for that unit is permitted. If the FILE= specifier is not included in such an OPEN statement, the file to be connected to the unit is the same as the file to which the unit is already connected.

6 If the file to be connected to the unit is the same as the file to which the unit is connected, a new connection is not established and values for any changeable modes (12.5.2) specified come into effect for the established connection; the current file position is unaffected. Before any effect on changeable modes, a wait operation is performed for any pending asynchronous data transfer operations for the specified unit. If the POSITION= specifier appears in such an OPEN statement, the value specified shall not disagree with the current position of the file. If the STATUS= specifier is included in such an OPEN statement, it shall be specified with the value OLD. Other than ERR=, IOSTAT=, and IOMSG=, and the changeable modes, the values of all other specifiers in such an OPEN statement shall not differ from those in effect for the established connection.

7 A STATUS= specifier with a value of OLD is always allowed when the file to be connected to the unit is the same as the file to which the unit is connected. In this case, if the status of the file was SCRATCH before execution of the OPEN statement, the file will still be deleted when the unit is closed, and the file is still considered to have a status of SCRATCH.

12.5.6.2 Syntax

R1204 open-stmt

is OPEN ( connect-spec-list )

R1205 connect-spec

is [ UNIT = ] file-unit-number
or ACCESS = scalar-default-char-expr
or ACTION = scalar-default-char-expr
or ASYNCHRONOUS = scalar-default-char-expr
or BLANK = scalar-default-char-expr
or DECIMAL = scalar-default-char-expr
or DELIM = scalar-default-char-expr
or ENCODING = scalar-default-char-expr
or ERR = label
or FILE = file-name-expr
or FORM = scalar-default-char-expr
or IOMSG = iomsg-variable
or IOSTAT = scalar-int-variable
or NEWUNIT = scalar-int-variable
or PAD = scalar-default-char-expr
or POSITION = scalar-default-char-expr
or RECL = scalar-int-expr
or ROUND = scalar-default-char-expr
or SIGN = scalar-default-char-expr
or STATUS = scalar-default-char-expr

R1206 file-name-expr is scalar-default-char-expr
R1207 iomsg-variable is scalar-default-char-variable

C1203 No specifier shall appear more than once in a given connect-spec-list.

C1204 (R1204) If the NEWUNIT= specifier does not appear, a file-unit-number shall be specified; if the optional characters UNIT= are omitted, the file-unit-number shall be the first item in the connect-spec-list.

C1205 (R1204) If a NEWUNIT= specifier appears, a file-unit-number shall not appear.

C1206 (R1204) The label used in the ERR= specifier shall be the statement label of a branch target statement that appears in the same inclusive scope as the OPEN statement.

1 If the STATUS= specifier has the value NEW or REPLACE, the FILE= specifier shall appear. If the STATUS= specifier has the value SCRATCH, the FILE= specifier shall not appear. If the STATUS= specifier has the value OLD, the FILE= specifier shall appear unless the unit is connected and the file connected to the unit exists.

2 If the NEWUNIT= specifier appears in an OPEN statement, either the FILE= specifier shall appear, or the STATUS= specifier shall appear with a value of SCRATCH. The unit identified by a NEWUNIT value shall not be preconnected.

3 A specifier that requires a scalar-default-char-expr may have a limited list of character values. These values are listed for each such specifier. Any trailing blanks are ignored. The value specified is without regard to case. Some specifiers have a default value if the specifier is omitted.

4 The IOSTAT=, ERR=, and IOMSG= specifiers are described in 12.11.

NOTE 12.18
An example of an OPEN statement is:

OPEN (10, FILE = 'employee.names', ACTION = 'READ', PAD = 'YES')

NOTE 12.19
For more explanatory information on the OPEN statement, see C.7.3.
12.5.6.3 ACCESS= specifier in the OPEN statement

The scalar-default-char-expr shall evaluate to SEQUENTIAL, DIRECT, or STREAM. The ACCESS= specifier specifies the access method for the connection of the file as being sequential, direct, or stream. If this specifier is omitted, the default value is SEQUENTIAL. For an existing file, the specified access method shall be included in the set of allowed access methods for the file. For a new file, the processor creates the file with a set of allowed access methods that includes the specified method.

12.5.6.4 ACTION= specifier in the OPEN statement

The scalar-default-char-expr shall evaluate to READ, WRITE, or READWRITE. READ specifies that the WRITE, PRINT, and ENDFILE statements shall not refer to this connection. WRITE specifies that READ statements shall not refer to this connection. READWRITE permits any input/output statements to refer to this connection. If this specifier is omitted, the default value is processor dependent. If READWRITE is included in the set of allowable actions for a file, both READ and WRITE also shall be included in the set of allowed actions for that file. For an existing file, the specified action shall be included in the set of allowed actions for the file. For a new file, the processor creates the file with a set of allowed actions that includes the specified action.

12.5.6.5 ASYNCHRONOUS= specifier in the OPEN statement

The scalar-default-char-expr shall evaluate to YES or NO. If YES is specified, asynchronous input/output on the unit is allowed. If NO is specified, asynchronous input/output on the unit is not allowed. If this specifier is omitted, the default value is NO.

12.5.6.6 BLANK= specifier in the OPEN statement

The scalar-default-char-expr shall evaluate to NULL or ZERO. The BLANK= specifier is permitted only for a connection for formatted input/output. It specifies the blank interpretation mode (13.8.6, 12.6.2.6) for input for this connection. This mode has no effect on output. It is a changeable mode (12.5.2). If this specifier is omitted in an OPEN statement that initiates a connection, the default value is NULL.

12.5.6.7 DECIMAL= specifier in the OPEN statement

The scalar-default-char-expr shall evaluate to COMMA or POINT. The DECIMAL= specifier is permitted only for a connection for formatted input/output. It specifies the decimal edit mode (13.6, 13.8.8, 12.6.2.7) for this connection. It is a changeable mode (12.5.2). If this specifier is omitted in an OPEN statement that initiates a connection, the default value is POINT.

12.5.6.8 DELIM= specifier in the OPEN statement

The scalar-default-char-expr shall evaluate to APOSTROPHE, QUOTE, or NONE. The DELIM= specifier is permitted only for a connection for list-directed (13.10.4) and namelist (13.11.4.2) output for the connection. This mode has no effect on input. It is a changeable mode (12.5.2). If this specifier is omitted in an OPEN statement that initiates a connection, the default value is NONE.

12.5.6.9 ENCODING= specifier in the OPEN statement

The scalar-default-char-expr shall evaluate to UTF-8 or DEFAULT. The ENCODING= specifier is permitted only for a connection for formatted input/output. It specifies the encoding form of the file is UTF-8 as specified in ISO/IEC 10646. Such a file is called a Unicode file, and all characters therein are of ISO 10646 character kind. The value UTF-8 shall not be specified if the processor does not support the ISO 10646 character kind. The value DEFAULT specifies that the encoding form of the file is processor dependent. If this specifier is omitted in an OPEN statement that initiates a connection, the default value is DEFAULT.
12.5.6.10  FILE= specifier in the OPEN statement

1 The value of the FILE= specifier is the name of the file to be connected to the specified unit. Any trailing blanks are ignored. The file-name-expr shall be a name that is allowed by the processor. If this specifier is omitted and the unit is not connected to a file, the STATUS= specifier shall be specified with a value of SCRATCH; in this case, the connection is made to a processor-dependent file. The interpretation of case is processor dependent.

12.5.6.11  FORM= specifier in the OPEN statement

1 The scalar-default-char-expr shall evaluate to FORMATTED or UNFORMATTED. The FORM= specifier determines whether the file is being connected for formatted or unformatted input/output. If this specifier is omitted, the default value is UNFORMATTED if the file is being connected for direct access or stream access, and the default value is FORMATTED if the file is being connected for sequential access. For an existing file, the specified form shall be included in the set of allowed forms for the file. For a new file, the processor creates the file with a set of allowed forms that includes the specified form.

12.5.6.12  NEWUNIT= specifier in the OPEN statement

1 The variable is defined with a processor determined NEWUNIT value if no error occurs during the execution of the OPEN statement. If an error occurs, the processor shall not change the value of the variable.

2 A NEWUNIT value is a negative number, and shall not be equal to −1, any of the named constants ERROR_UNIT, INPUT_UNIT, or OUTPUT_UNIT from the intrinsic module ISO_FORTRAN_ENV (16.10.2), any value used by the processor for the unit argument to a defined input/output procedure, nor any previous NEWUNIT value that identifies a file that is connected.

12.5.6.13  PAD= specifier in the OPEN statement

1 The scalar-default-char-expr shall evaluate to YES or NO. The PAD= specifier is permitted only for a connection for formatted input/output. It specifies the pad mode (12.6.4.5.3, 12.6.2.10) for input for this connection. This mode has no effect on output. It is a changeable mode (12.5.2). If this specifier is omitted in an OPEN statement that initiates a connection, the default value is YES.

12.5.6.14  POSITION= specifier in the OPEN statement

1 The scalar-default-char-expr shall evaluate to ASIS, REWIND, or APPEND. The connection shall be for sequential or stream access. A new file is positioned at its initial point. REWIND positions an existing file at its initial point. APPEND positions an existing file such that the endfile record is the next record, if it has one. If an existing file does not have an endfile record, APPEND positions the file at its terminal point. ASIS leaves the position unchanged if the file exists and already is connected. If the file exists but is not connected, the position resulting from ASIS is processor dependent. If this specifier is omitted, the default value is ASIS.

12.5.6.15  RECL= specifier in the OPEN statement

1 The value of the RECL= specifier shall be positive. It specifies the length of each record in a file being connected for direct access, or specifies the maximum length of a record in a file being connected for sequential access. This specifier shall not appear when a file is being connected for stream access. This specifier shall appear when a file is being connected for direct access. If this specifier is omitted when a file is being connected for sequential access, the default value is processor dependent. If the file is being connected for formatted input/output, the length is the number of characters for all records that contain only characters of default kind. When a record contains any nondefault characters, the effect of the RECL= specifier is processor dependent. If the file is being connected for unformatted input/output, the length is measured in file storage units. For an existing file, the value of the RECL= specifier shall be included in the set of allowed record lengths for the file. For a new file, the processor creates the file with a set of allowed record lengths that includes the specified value.
12.5.6.16 ROUND= specifier in the OPEN statement

The scalar-default-char-expr shall evaluate to one of UP, DOWN, ZERO, NEAREST, COMPATIBLE, or PROCESSOR_DEFINED. The ROUND= specifier is permitted only for a connection for formatted input/output. It specifies the input/output rounding mode (13.7.2.3.8, 12.6.2.13) for this connection. It is a changeable mode (12.5.2). If this specifier is omitted in an OPEN statement that initiates a connection, the input/output rounding mode is processor dependent; it shall be one of the above modes.

NOTE 12.20

A processor is free to select any input/output rounding mode for the default mode. The mode might correspond to UP, DOWN, ZERO, NEAREST, or COMPATIBLE; or it might be a completely different input/output rounding mode.

12.5.6.17 SIGN= specifier in the OPEN statement

The scalar-default-char-expr shall evaluate to one of PLUS, SUPPRESS, or PROCESSOR_DEFINED. The SIGN= specifier is permitted only for a connection for formatted input/output. It specifies the sign mode (13.8.4, 12.6.2.14) for this connection. It is a changeable mode (12.5.2). If this specifier is omitted in an OPEN statement that initiates a connection, the default value is PROCESSOR_DEFINED.

12.5.6.18 STATUS= specifier in the OPEN statement

The scalar-default-char-expr shall evaluate to OLD, NEW, SCRATCH, REPLACE, or UNKNOWN. If OLD is specified, the file shall exist. If NEW is specified, the file shall not exist.

Successful execution of an OPEN statement with NEW specified creates the file and changes the status to OLD. If REPLACE is specified and the file does not already exist, the file is created and the status is changed to OLD. If REPLACE is specified and the file does exist, the file is deleted, a new file is created with the same name, and the status is changed to OLD. If SCRATCH is specified, the file is created and connected to the specified unit for use by the program but is deleted at the execution of a CLOSE statement referring to the same unit or at the normal termination of the program. SCRATCH shall not be specified if the FILE= specifier appears.

If UNKNOWN is specified, the status is processor dependent. If this specifier is omitted, the default value is UNKNOWN.

12.5.7 CLOSE statement

12.5.7.1 General

The CLOSE statement is used to terminate the connection of a specified unit to an external file.

Execution of a CLOSE statement for a unit may occur in any program unit of a program and need not occur in the same program unit as the execution of an OPEN statement referring to that unit.

Execution of a CLOSE statement performs a wait operation for any pending asynchronous data transfer operations for the specified unit.

Execution of a CLOSE statement specifying a unit that does not exist, exists but is connected to a file that does not exist, or has no file connected to it, is permitted and affects no file or unit.

After a unit has been disconnected by execution of a CLOSE statement, it may be connected again within the same program, either to the same file or to a different file. After a named file has been disconnected by execution of a CLOSE statement, it may be connected again within the same program, either to the same unit or to a different unit, provided that the file still exists.

During the completion step (5.3.7) of normal termination, all units that are connected are closed. Each unit is closed with status KEEP unless the file status prior to termination of execution was SCRATCH, in which case
the unit is closed with status DELETE.

**NOTE 12.21**
The effect is as though a CLOSE statement without a STATUS= specifier were executed on each connected unit.

### 12.5.7.2 Syntax

**R1208** close-stmt

is

\[
\text{CLOSE ( close-spec-list )}
\]

**R1209** close-spec

is

\[
\begin{align*}
&\text{[ UNIT = ] file-unit-number} \\
&\text{or IOSTAT = scalar-int-variable} \\
&\text{or IOMSG = iomsg-variable} \\
&\text{or ERR = label} \\
&\text{or STATUS = scalar-default-char-expr}
\end{align*}
\]

C1207 No specifier shall appear more than once in a given close-spec-list.

C1208 A file-unit-number shall be specified in a close-spec-list; if the optional characters UNIT= are omitted, the file-unit-number shall be the first item in the close-spec-list.

C1209 (R1209) The label used in the ERR= specifier shall be the statement label of a branch target statement that appears in the same inclusive scope as the CLOSE statement.

1. The scalar-default-char-expr has a limited list of character values. Any trailing blanks are ignored. The value specified is without regard to case.

2. The IOSTAT=, ERR=, and IOMSG= specifiers are described in 12.11.

**NOTE 12.22**

An example of a CLOSE statement is:

\[
\text{CLOSE (10, STATUS = 'KEEP')}
\]

### 12.5.7.3 STATUS= specifier in the CLOSE statement

1. The scalar-default-char-expr shall evaluate to KEEP or DELETE. The STATUS= specifier determines the disposition of the file that is connected to the specified unit. KEEP shall not be specified for a file whose status prior to execution of a CLOSE statement is SCRATCH. If KEEP is specified for a file that exists, the file continues to exist after the execution of a CLOSE statement. If KEEP is specified for a file that does not exist, the file will not exist after the execution of a CLOSE statement. If DELETE is specified, the file will not exist after the execution of a CLOSE statement. If this specifier is omitted, the default value is KEEP, unless the file status prior to execution of the CLOSE statement is SCRATCH, in which case the default value is DELETE.

### 12.6 Data transfer statements

#### 12.6.1 Form of input and output statements

1. The READ statement is the data transfer input statement. The WRITE statement and the PRINT statement are the data transfer output statements.
**NOTE 12.23**

Examples of data transfer statements are:

```plaintext
READ (6, *) SIZE
READ 10, A, B
WRITE (6, 10) A, S, J
PRINT 10, A, S, J
10 FORMAT (2E16.3, I5)
```

## 12.6.2 Control information list

### 12.6.2.1 Syntax

A control information list is an `io-control-spec-list`. It governs data transfer.

**R1213**

```plaintext
io-control-spec is [ UNIT = ] io-unit
or [ FMT = ] format
or [ NML = ] namelist-group-name
or ADVANCE = scalar-default-char-expr
or ASYNCHRONOUS = scalar-default-char-constant-expr
or BLANK = scalar-default-char-expr
or DECIMAL = scalar-default-char-expr
or DELIM = scalar-default-char-expr
or END = label
or EOR = label
or ERR = label
or ID = id-variable
or IOMSG = iomsg-variable
or IOSTAT = scalar-int-variable
or PAD = scalar-default-char-expr
or POS = scalar-int-expr
or REC = scalar-int-expr
or ROUND = scalar-default-char-expr
or SIGN = scalar-default-char-expr
or SIZE = scalar-int-variable
```

**R1214**

```plaintext
id-variable is scalar-int-variable
```

**C1210** No specifier shall appear more than once in a given `io-control-spec-list`.

**C1211** An `io-unit` shall be specified in an `io-control-spec-list`; if the optional characters UNIT= are omitted, the `io-unit` shall be the first item in the `io-control-spec-list`.

**C1212** (R1213) A DELIM= or SIGN= specifier shall not appear in a `read-stmt`.

**C1213** (R1213) A BLANK=, PAD=, END=, EOR=, or SIZE= specifier shall not appear in a `write-stmt`.

**C1214** (R1213) The label in the ERR=, EOR=, or END= specifier shall be the statement label of a branch target statement that appears in the same inclusive scope as the data transfer statement.

**C1215** (R1213) A `namelist-group-name` shall be the name of a namelist group.

**C1216** (R1213) A `namelist-group-name` shall not appear if a REC= specifier, format, input-item-list, or an output-item-list appears in the data transfer statement.

**C1217** (R1213) If format appears without a preceding FMT=, it shall be the second item in the `io-control-spec-list`.
list and the first item shall be \textit{io-unit}.

C1218 (R1213) If \textit{namelist-group-name} appears without a preceding NML=, it shall be the second item in the \textit{io-control-spec-list} and the first item shall be \textit{io-unit}.

C1219 (R1213) If \textit{io-unit} is not a \textit{file-unit-number}, the \textit{io-control-spec-list} shall not contain a REC= specifier or a POS= specifier.

C1220 (R1213) If the REC= specifier appears, an END= specifier shall not appear, and the \textit{format}, if any, shall not be an asterisk.

C1221 (R1213) An ADVANCE= specifier may appear only in a formatted sequential or stream data transfer statement with explicit format specification (13.2) whose \textit{io-control-spec-list} does not contain an \textit{internal-file-variable} as the \textit{io-unit}.

C1222 (R1213) If an EOR= specifier appears, an ADVANCE= specifier also shall appear.

C1223 (R1213) The \textit{scalar-default-char-constant-expr} in an ASYNCHRONOUS= specifier shall have the value YES or NO.

C1224 (R1213) An ASYNCHRONOUS= specifier with a value YES shall not appear unless \textit{io-unit} is a \textit{file-unit-number}.

C1225 (R1213) If an ID= specifier appears, an ASYNCHRONOUS= specifier with the value YES shall also appear.

C1226 (R1213) If a POS= specifier appears, the \textit{io-control-spec-list} shall not contain a REC= specifier.

C1227 (R1213) If a DECIMAL=, BLANK=, PAD=, SIGN=, or ROUND= specifier appears, a \textit{format} or \textit{namelist-group-name} shall also appear.

C1228 (R1213) If a DELIM= specifier appears, either \textit{format} shall be an asterisk or \textit{namelist-group-name} shall appear.

C1229 (R1214) The \textit{scalar-int-variable} shall have a decimal exponent range no smaller than that of default integer.

If an EOR= specifier appears, an ADVANCE= specifier with the value NO shall also appear.

If the data transfer statement contains a \textit{format} or \textit{namelist-group-name}, the statement is a formatted input/output statement; otherwise, it is an unformatted input/output statement.

The ADVANCE=, ASYNCHRONOUS=, DECIMAL=, BLANK=, PAD=, SIGN=, and ROUND= specifiers have a limited list of character values. Any trailing blanks are ignored. The values specified are without regard to case.

The IOSTAT=, ERR=, EOR=, END=, and IOMSG= specifiers are described in 12.11.

\textbf{NOTE 12.24}

An example of a READ statement is:

\begin{verbatim}
READ (IOSTAT = IOS, UNIT = 6, FMT = '(10F8.2)') A, B
\end{verbatim}

\textbf{12.6.2.2 Format specification in a data transfer statement}

The \textit{format} specifier supplies a format specification or specifies list-directed formatting for a formatted input/output statement.
R1215  format is default-char-expr or label or *

C1230  (R1215) The label shall be the label of a FORMAT statement that appears in the same inclusive scope as the statement containing the FMT= specifier.

2 The default-char-expr shall evaluate to a valid format specification (13.2.1 and 13.2.2).

3 If default-char-expr is an array, it is treated as if all of the elements of the array were specified in array element order and were concatenated.

4 If format is *, the statement is a list-directed input/output statement.

NOTE 12.25
An example in which the format is a character expression is:

```
READ (6, FMT = "(" // CHAR_FMT // ")") X, Y, Z
```

where CHAR_FMT is a default character variable.

12.6.2.3 NML= specifier in a data transfer statement

1 The NML= specifier supplies the namelist-group-name (8.9). This name identifies a particular collection of data objects on which transfer is to be performed.

2 If a namelist-group-name appears, the statement is a namelist input/output statement.

12.6.2.4 ADVANCE= specifier in a data transfer statement

1 The scalar-default-char-expr shall evaluate to YES or NO. The ADVANCE= specifier determines whether advancing input/output occurs for a nonchild data transfer statement. If YES is specified for a nonchild data transfer statement, advancing input/output occurs. If NO is specified, nonadvancing input/output occurs (12.3.4.2). If this specifier is omitted from a nonchild data transfer statement that allows the specifier, the default value is YES. A formatted child data transfer statement is a nonadvancing input/output statement, and any ADVANCE= specifier is ignored.

12.6.2.5 ASYNCHRONOUS= specifier in a data transfer statement

1 The ASYNCHRONOUS= specifier determines whether this data transfer statement is synchronous or asynchronous. If YES is specified, the statement and the input/output operation are asynchronous. If NO is specified or if the specifier is omitted, the statement and the input/output operation are synchronous.

2 Asynchronous input/output is permitted only for external files opened with an ASYNCHRONOUS= specifier with the value YES in the OPEN statement.

NOTE 12.26

Both synchronous and asynchronous input/output are allowed for files opened with an ASYNCHRONOUS= specifier of YES. For other files, only synchronous input/output is allowed; this includes files opened with an ASYNCHRONOUS= specifier of NO, files opened without an ASYNCHRONOUS= specifier, preconnected files accessed without an OPEN statement, and internal files.

The ASYNCHRONOUS= specifier value in a data transfer statement is a constant expression because it effects compiler optimizations and, therefore, needs to be known at compile time.

3 The processor may perform an asynchronous data transfer operation asynchronously, but it is not required to do so. For each external file, records and file storage units read or written by asynchronous data transfer statements
are read, written, and processed in the same order as they would have been if the data transfer statements were synchronous. The documentation of the Fortran processor should describe when input/output will be performed asynchronously.

If a variable is used in an asynchronous data transfer statement as

- an item in an input/output list,
- a group object in a namelist, or
- a SIZE= specifier,

the base object of the data-ref is implicitly given the ASYNCHRONOUS attribute in the scoping unit of the data transfer statement. This attribute may be confirmed by explicit declaration.

When an asynchronous input/output statement is executed, the set of storage units specified by the item list or NML= specifier, plus the storage units specified by the SIZE= specifier, is defined to be the pending input/output storage sequence for the data transfer operation.

NOTE 12.27

A pending input/output storage sequence is not necessarily a contiguous set of storage units.

A pending input/output storage sequence affector is a variable of which any part is associated with a storage unit in a pending input/output storage sequence.

12.6.2.6 BLANK= specifier in a data transfer statement

The scalar-default-char-expr shall evaluate to NULL or ZERO. The BLANK= specifier temporarily changes (12.5.2) the blank interpretation mode (13.8.6, 12.5.6.6) for the connection. If the specifier is omitted, the mode is not changed.

12.6.2.7 DECIMAL= specifier in a data transfer statement

The scalar-default-char-expr shall evaluate to COMMA or POINT. The DECIMAL= specifier temporarily changes (12.5.2) the decimal edit mode (13.6, 13.8.8, 12.5.6.7) for the connection. If the specifier is omitted, the mode is not changed.

12.6.2.8 DELIM= specifier in a data transfer statement

The scalar-default-char-expr shall evaluate to APOSTROPHE, QUOTE, or NONE. The DELIM= specifier temporarily changes (12.5.2) the delimiter mode (13.10.4, 13.11.4.2, 12.5.6.8) for the connection. If the specifier is omitted, the mode is not changed.

12.6.2.9 ID= specifier in a data transfer statement

Successful execution of an asynchronous data transfer statement containing an ID= specifier causes the variable specified in the ID= specifier to become defined with a processor determined value. If this value is zero, the data transfer operation has been completed. A nonzero value is referred to as the identifier of the data transfer operation. This identifier is different from the identifier of any other pending data transfer operation for this unit. It can be used in a subsequent WAIT or INQUIRE statement to identify the particular data transfer operation. If an error occurs during the execution of a data transfer statement containing an ID= specifier, the variable specified in the ID= specifier becomes undefined.

A child data transfer statement shall not specify the ID= specifier.
12.6.2.10 PAD= specifier in a data transfer statement

The scalar-default-char-expr shall evaluate to YES or NO. The PAD= specifier temporarily changes (12.5.2) the pad mode (12.6.4.5.3, 12.5.6.13) for the connection. If the specifier is omitted, the mode is not changed.

12.6.2.11 POS= specifier in a data transfer statement

The POS= specifier specifies the file position in file storage units. This specifier may appear in a data transfer statement only if the statement specifies a unit connected for stream access. A child data transfer statement shall not specify this specifier.

A processor may prohibit the use of POS= with particular files that do not have the properties necessary to support random positioning. A processor may also prohibit positioning a particular file to any position prior to its current file position if the file does not have the properties necessary to support such positioning.

NOTE 12.28
A unit that is connected to a device or data stream might not be positionable.

If the file is connected for formatted stream access, the file position specified by POS= shall be equal to either 1 (the beginning of the file) or a value previously returned by a POS= specifier in an INQUIRE statement for the file.

12.6.2.12 REC= specifier in a data transfer statement

The REC= specifier specifies the number of the record that is to be read or written. This specifier may appear only in a data transfer statement that specifies a unit connected for direct access; it shall not appear in a child data transfer statement. If the io-control-spec-list contains a REC= specifier, the statement is a direct access data transfer statement. A child data transfer statement is a direct access data transfer statement if the parent is a direct access data transfer statement. Any other data transfer statement is a sequential access data transfer statement or a stream access data transfer statement, depending on whether the file connection is for sequential access or stream access.

12.6.2.13 ROUND= specifier in a data transfer statement

The scalar-default-char-expr shall evaluate to UP, DOWN, ZERO, NEAREST, COMPATIBLE or PROCESSOR_DEFINED. The ROUND= specifier temporarily changes (12.5.2) the input/output rounding mode (13.7.2.3.8, 12.5.6.16) for the connection. If the specifier is omitted, the mode is not changed.

12.6.2.14 SIGN= specifier in a data transfer statement

The scalar-default-char-expr shall evaluate to PLUS, SUPPRESS, or PROCESSOR_DEFINED. The SIGN= specifier temporarily changes (12.5.2) the sign mode (13.8.4, 12.5.6.17) for the connection. If the specifier is omitted, the mode is not changed.

12.6.2.15 SIZE= specifier in a data transfer statement

The SIZE= specifier in an input statement causes the variable specified to become defined with the count of the characters transferred from the file by data edit descriptors during the input operation. Blanks inserted as padding are not counted.

For a synchronous input statement, this definition occurs when execution of the statement completes. For an asynchronous input statement, this definition occurs when the corresponding wait operation is performed.

12.6.3 Data transfer input/output list

An input/output list specifies the entities whose values are transferred by a data transfer statement.
R1216  input-item is variable
        or io-implied-do

R1217  output-item is expr
        or io-implied-do

R1218  io-implied-do is ( io-implied-do-object-list , io-implied-do-control )

R1219  io-implied-do-object is input-item
        or output-item

R1220  io-implied-do-control is do-variable = scalar-int-expr ,

        scalar-int-expr [ , scalar-int-expr ]

C1231  (R1216) A variable that is an input-item shall not be a whole assumed-size array.

C1232  (R1219) In an input-item-list, an io-implied-do-object shall be an input-item. In an output-item-list, an

        io-implied-do-object shall be an output-item.

C1233  (R1217) An expression that is an output-item shall not have a value that is a procedure pointer.

An input-item shall not appear as, nor be associated with, the do-variable of any io-implied-do that contains the
input-item.

NOTE 12.29
A constant, an expression involving operators or function references that does not have a pointer result, or
an expression enclosed in parentheses shall not appear as an input list item.

3 If an input item is a pointer, it shall be associated with a definable target and data are transferred from the file to
the associated target. If an output item is a pointer, it shall be associated with a target and data are transferred
from the target to the file.

NOTE 12.30
Data transfers always involve the movement of values between a file and internal storage. A pointer as such
cannot be read or written. Therefore, a pointer shall not appear as an item in an input/output list unless
it is associated with a target that can receive a value (input) or can deliver a value (output).

4 If an input item or an output item is allocatable, it shall be allocated.

5 A list item shall not be polymorphic unless it is processed by a defined input/output procedure (12.6.4.8).

6 The do-variable of an io-implied-do that is in another io-implied-do shall not appear as, nor be associated with,
the do-variable of the containing io-implied-do.

7 The following rules describing whether to expand an input/output list item are re-applied to each expanded list
item until none of the rules apply.

• If an array appears as an input/output list item, it is treated as if the elements, if any, were specified in
array element order (9.5.3.2). However, no element of that array may affect the value of any expression in
the input-item, nor may any element appear more than once in an input-item.

NOTE 12.31
For example:

INTEGER A (100), J (100)

... READ *, A (A) ! Not allowed
READ *, A (LBOUND (A, 1) : UBOUND (A, 1)) ! Allowed
NOTE 12.31 (cont.)

<table>
<thead>
<tr>
<th>READ *, A (J)</th>
<th>! Allowed if no two elements of J have the same value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(1) = 1; A(10) = 10</td>
<td>!</td>
</tr>
<tr>
<td>READ *, A (A(1) : A(10))</td>
<td>! Not allowed</td>
</tr>
</tbody>
</table>

• If a list item of derived type in an unformatted input/output statement is not processed by a defined input/output procedure (12.6.4.8), and if any subobject of that list item would be processed by a defined input/output procedure, the list item is treated as if all of the components of the object were specified in the list in component order (7.5.4.7); those components shall be accessible in the scoping unit containing the data transfer statement and shall not be pointers or allocatable.

• An effective item of derived type in an unformatted input/output statement is treated as a single value in a processor-dependent form unless the list item or a subobject thereof is processed by a defined input/output procedure (12.6.4.8).

NOTE 12.32

The appearance of a derived-type object as an input/output list item in an unformatted input/output statement is not equivalent to the list of its components.

Unformatted input/output involving derived-type list items forms the single exception to the rule that the appearance of an aggregate list item (such as an array) is equivalent to the appearance of its expanded list of component parts. This exception permits the processor greater latitude in improving efficiency or in matching the processor-dependent sequence of values for a derived-type object to similar sequences for aggregate objects used by means other than Fortran. However, formatted input/output of all list items and unformatted input/output of list items other than those of derived types adhere to the above rule.

• If a list item of derived type in a formatted input/output statement is not processed by a defined input/output procedure, that list item is treated as if all of the components of the list item were specified in the list in component order; those components shall be accessible in the scoping unit containing the input/output statement and shall not be pointers or allocatable.

• If a derived-type list item is not processed by a defined input/output procedure and is not treated as a list of its individual components, all the subcomponents of that list item shall be accessible in the scoping unit containing the data transfer statement and shall not be pointers or allocatable.

• For an io-implied-do, the loop initialization and execution are the same as for a DO construct (11.1.7.4).

NOTE 12.33

An example of an output list with an implied DO is:

```
WRITE (LP, FMT = '(10F8.2)') (LOG (A(I)), I = 1, N + 9, K), G
```

The scalar objects resulting when a data transfer statement’s list items are expanded according to the rules in this subclause for handling array and derived-type list items are called effective items. Zero-sized arrays and io-implied-dos with an iteration count of zero do not contribute to the list of effective items. A scalar character item of zero length is an effective item.

NOTE 12.34

In a formatted input/output statement, edit descriptors are associated with effective items, which are always scalar. The rules in 12.6.3 determine the set of effective items corresponding to each actual list item in the statement. These rules might have to be applied repetitively until all of the effective items are scalar items.

An input/output list shall not contain an effective item of nondefault character kind if the data transfer statement specifies an internal file of default character kind. An input/output list shall not contain an effective item that is nondefault character except for ISO 10646 or ASCII character if the data transfer statement specifies an internal file of ISO 10646 character kind. An input/output list shall not contain an effective item of type character of any kind other than ASCII if the data transfer statement specifies an ASCII character internal file.
12.6.4 Execution of a data transfer input/output statement

12.6.4.1 Data transfer sequence of operations

1 Execution of a WRITE or PRINT statement for a unit connected to a file that does not exist creates the file unless an error condition occurs.

2 The effect of executing a synchronous data transfer statement shall be as if the following operations were performed in the order specified.

   (1) Determine the direction of data transfer (12.6.4.2).
   (2) Identify the unit (12.6.4.3).
   (3) Perform a wait operation for all pending input/output operations for the unit. If an error, end-of-file, or end-of-record condition occurs during any of the wait operations, steps 4 through 8 are skipped.
   (4) Establish the format if one is specified.
   (5) If the statement is not a child data transfer statement (12.6.4.8),
      (a) position the file prior to data transfer (12.3.4.3), and
      (b) for formatted data transfer, set the left tab limit (13.8.1.2).
   (6) Transfer data between the file and the entities specified by the input/output list (if any) or namelist, possibly mediated by defined input/output procedures (12.6.4.8).
   (7) Determine whether an error, end-of-file, or end-of-record condition has occurred.
   (8) Position the file after data transfer (12.3.4.4) unless the statement is a child data transfer statement (12.6.4.8).
   (9) Cause any variable specified in a SIZE= specifier to become defined.
   (10) If an error, end-of-file, or end-of-record condition occurred, processing continues as specified in 12.11; otherwise, any variable specified in an IOSTAT= specifier is assigned the value zero.

3 The effect of executing an asynchronous data transfer statement shall be as if the following operations were performed in the order specified.

   (1) Determine the direction of data transfer (12.6.4.2).
   (2) Identify the unit (12.6.4.3).
   (3) Optionally, perform wait operations for one or more pending input/output operations for the unit. If an error, end-of-file, or end-of-record condition occurs during any of the wait operations, steps 4 through 9 are skipped.
   (4) Establish the format if one is specified.
   (5) Position the file prior to data transfer (12.3.4.3) and, for formatted data transfer, set the left tab limit (13.8.1.2).
   (6) Establish the set of storage units identified by the input/output list. For an input statement, this might require some or all of the data in the file to be read if an input variable is used as a scalar-int-expr in an io-implied-do-control in the input/output list, as a subscript, substring-range, stride, or is otherwise referenced.
   (7) Initiate an asynchronous data transfer between the file and the entities specified by the input/output list (if any) or namelist. The asynchronous data transfer may complete (and an error, end-of-file, or end-of-record condition may occur) during the execution of this data transfer statement or during a later wait operation.
   (8) Determine whether an error, end-of-file, or end-of-record condition has occurred. The conditions may occur during the execution of this data transfer statement or during the corresponding wait operation, but not both.
   (9) Position the file as if the data transfer had finished (12.3.4.4).
   (10) Cause any variable specified in a SIZE= specifier to become undefined.
   (11) If an error, end-of-file, or end-of-record condition occurred, processing continues as specified in 12.11; otherwise, any variable specified in an IOSTAT= specifier is assigned the value zero.
For an asynchronous data transfer statement, the data transfers may occur during execution of the statement, during execution of the corresponding wait operation, or anywhere between. The data transfer operation is considered to be pending until a corresponding wait operation is performed.

For asynchronous output, a pending input/output storage sequence affector (12.6.2.5) shall not be redefined, become undefined, or have its pointer association status changed.

For asynchronous input, a pending input/output storage sequence affector shall not be referenced, become defined, become undefined, become associated with a dummy argument that has the VALUE attribute, or have its pointer association status changed.

Error, end-of-file, and end-of-record conditions in an asynchronous data transfer operation may occur during execution of either the data transfer statement or the corresponding wait operation. If an ID= specifier does not appear in the initiating data transfer statement, the conditions may occur during the execution of any subsequent data transfer or wait operation for the same unit. When a condition occurs for a previously executed asynchronous data transfer statement, a wait operation is performed for all pending data transfer operations on that unit. When a condition occurs during a subsequent statement, any actions specified by IOSTAT=, IOMSG=, ERR=, END=, and EOR= specifiers for that statement are taken.

NOTE 12.35
Because end-of-file and error conditions for asynchronous data transfer statements without an ID= specifier can be reported by the processor during the execution of a subsequent data transfer statement, it might be impossible for the user to determine which data transfer statement caused the condition. Reliably detecting which input statement caused an end-of-file condition requires that all asynchronous input statements for the unit include an ID= specifier.

12.6.4.2 Direction of data transfer

Execution of a READ statement causes values to be transferred from a file to the entities specified by the input list, if any, or specified within the file itself for namelist input. Execution of a WRITE or PRINT statement causes values to be transferred to a file from the entities specified by the output list and format specification, if any, or by the namelist-group-name for namelist output.

12.6.4.3 Identifying a unit

A data transfer statement that contains an input/output control list includes a UNIT= specifier that identifies an external or internal unit. A READ statement that does not contain an input/output control list specifies a particular processor-dependent unit, which is the same as the unit identified by * in a READ statement that contains an input/output control list (12.5.1) and is the same as the unit identified by the value of the named constant INPUT_UNIT of the intrinsic module ISO_FORTRAN_ENV (16.10.2.13). The PRINT statement specifies some other processor-dependent unit, which is the same as the unit identified by * in a WRITE statement and is the same as the unit identified by the value of the named constant OUTPUT_UNIT of the intrinsic module ISO_FORTRAN_ENV (16.10.2.22). Thus, each data transfer statement identifies an external or internal unit.

The unit identified by an unformatted data transfer statement shall be an external unit.

The unit identified by a data transfer statement shall be connected to a file when execution of the statement begins.

NOTE 12.36
The unit could be preconnected.
12.6.4.4 Establishing a format

1 If the input/output control list contains * as a format, list-directed formatting is established. If namelist-group-name appears, namelist formatting is established. If no format or namelist-group-name is specified, unformatted data transfer is established. Otherwise, the format specified by format is established.

2 For output to an internal file, a format specification that is in the file or is associated with the file shall not be specified.

3 An input list item, or an entity associated with it, shall not contain any portion of an established format specification.

12.6.4.5 Data transfer

12.6.4.5.1 General

1 Data are transferred between the file and the entities specified by the input/output list or namelist. The list items are processed in the order of the input/output list for all data transfer statements except namelist data transfer statements. The list items for a namelist input statement are processed in the order of the entities specified within the input records. The list items for a namelist output statement are processed in the order in which the variables are specified in the namelist-group-object-list. Effective items are derived from the input/output list items as described in 12.6.3.

2 All values needed to determine which entities are specified by an input/output list item are determined at the beginning of the processing of that item.

3 All values are transmitted to or from the entities specified by a list item prior to the processing of any succeeding list item for all data transfer statements.

NOTE 12.37

In the example

```plaintext
READ (N) N, X (N)
```

the old value of N identifies the `unit`, but the new value of N is the subscript of X.

4 All values following the `name=` part of the namelist entity (13.11) within the input records are transmitted to the matching entity specified in the namelist-group-object-list prior to processing any succeeding entity within the input record for namelist input statements. If an entity is specified more than once within the input record during a namelist input statement, the last occurrence of the entity specifies the value or values to be used for that entity.

5 If the input/output item is a pointer, data are transferred between the file and the associated target.

6 If an internal file has been specified, an input/output list item shall not be in the file or associated with the file.

7 During the execution of an output statement that specifies an internal file, no part of that internal file shall be referenced, defined, or become undefined as the result of evaluating any output list item.

8 During the execution of an input statement that specifies an internal file, no part of that internal file shall be defined or become undefined as the result of transferring a value to any input list item.

9 A DO variable becomes defined and its iteration count established at the beginning of processing of the io-implied-do-object-list an io-implied-do.

10 On output, every entity whose value is to be transferred shall be defined.
12.6.4.5.2 Unformatted data transfer

If the file is not connected for unformatted input/output, unformatted data transfer is prohibited.

During unformatted data transfer, data are transferred without editing between the file and the entities specified by the input/output list. If the file is connected for sequential or direct access, exactly one record is read or written.

A value in the file is stored in a contiguous sequence of file storage units, beginning with the file storage unit immediately following the current file position.

After each value is transferred, the current file position is moved to a point immediately after the last file storage unit of the value.

On input from a file connected for sequential or direct access, the number of file storage units required by the input list shall be less than or equal to the number of file storage units in the record.

On input, if the file storage units transferred do not contain a value with the same type and type parameters as the input list entity, then the resulting value of the entity is processor dependent except in the following cases.

- A complex entity may correspond to two real values with the same kind type parameter as the complex entity.
- A default character list entity of length \( n \) may correspond to \( n \) default characters stored in the file, regardless of the length parameters of the entities that were written to these storage units of the file. If the file is connected for stream input, the characters may have been written by formatted stream output.

On output to a file connected for unformatted direct access, the output list shall not specify more values than can fit into the record. If the file is connected for direct access and the values specified by the output list do not fill the record, the remainder of the record is undefined.

If the file is connected for unformatted sequential access, the record is created with a length sufficient to hold the values from the output list. This length shall be one of the set of allowed record lengths for the file and shall not exceed the value specified in the RECL= specifier, if any, of the OPEN statement that established the connection.

12.6.4.5.3 Formatted data transfer

If the file is not connected for formatted input/output, formatted data transfer is prohibited.

During formatted data transfer, data are transferred with editing between the file and the entities specified by the input/output list or by the namelist-group-name. Format control is initiated and editing is performed as described in Clause 13.

The current record and possibly additional records are read or written.

During advancing input when the pad mode has the value NO, the input list and format specification shall not require more characters from the record than the record contains.

During advancing input when the pad mode has the value YES, blank characters are supplied by the processor if the input list and format specification require more characters from the record than the record contains.

During nonadvancing input when the pad mode 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, and the record is complete. If the record is incomplete, an end-of-file condition occurs instead of an end-of-record condition.

During nonadvancing input when the pad mode has the value YES, blank characters are supplied by the processor if an effective item and its corresponding data edit descriptors require more characters from the record than the record contains. If the record is incomplete, an end-of-field condition occurs; otherwise, an end-of-record condition occurs.
occurs.

If the file is connected for direct access, the record number is increased by one as each succeeding record is read
or written.

On output, if the file is connected for direct access or is an internal file and the characters specified by the output
list and format do not fill a record, blank characters are added to fill the record.

On output, the output list and format specification shall not specify more characters for a record than have been
specified by a RECL= specifier in the OPEN statement or the record length of an internal file.

12.6.4.6 List-directed formatting

1 If list-directed formatting has been established, editing is performed as described in 13.10.

12.6.4.7 Namelist formatting

1 If namelist formatting has been established, editing is performed as described in 13.11.

2 Every allocatable namelist-group-object in the namelist group shall be allocated and every namelist-group-object
that is a pointer shall be associated with a target. If a namelist-group-object is polymorphic or has an ultimate
component that is allocatable or a pointer, that object shall be processed by a defined input/output procedure
(12.6.4.8).

12.6.4.8 Defined input/output

12.6.4.8.1 General

1 Defined input/output allows a program to override the default handling of derived-type objects and values in
data transfer statements described in 12.6.3.

2 A defined input/output procedure is a procedure accessible by a defined-io-generic-spec (15.4.3.2). A particular
defined input/output procedure is selected as described in 12.6.4.8.4.

12.6.4.8.2 Defined input/output procedures

1 For a particular derived type and a particular set of kind type parameter values, there are four possible sets of
characteristics for defined input/output procedures; one each for formatted input, formatted output, unformatted
input, and unformatted output. The program need not supply all four procedures. The procedures are specified
to be used for derived-type input/output by interface blocks (15.4.3.2) or by generic bindings (7.5.5), with a
defined-io-generic-spec (R1509). The defined-io-generic-specs for these procedures are READ (FORMATTED),
READ (UNFORMATTED), WRITE (FORMATTED), and WRITE (UNFORMATTED), for formatted input,
umformatted input, formatted output, and unformatted output respectively.

2 In the four interfaces, which specify the characteristics of defined input/output procedures, the following syntax
term is used:

R1221 dte-type-spec is TYPE( derived-type-spec )
or CLASS( derived-type-spec )

C1234 (R1221) If derived-type-spec specifies an extensible type, the CLASS keyword shall be used; otherwise,
the TYPE keyword shall be used.

C1235 (R1221) All length type parameters of derived-type-spec shall be assumed.

3 If the defined-io-generic-spec is READ (FORMATTED), the characteristics shall be the same as those specified
by the following interface:

SUBROUTINE my_read_routine_formatted &
(dtv, &
unit, &
iotype, v_list, &
iostat, iomsg)

! the derived-type variable
dtv-type-spec, INTENT(INOUT) :: dtv
INTEGER, INTENT(IN) :: unit ! unit number

! the edit descriptor string
CHARACTER (LEN=__), INTENT(IN) :: iotype
INTEGER, INTENT(IN) :: v_list(:)
INTEGER, INTENT(OUT) :: iostat
CHARACTER (LEN=__), INTENT(INOUT) :: iomsg

END

5 If the defined-io-generic-spec is READ (UNFORMATTED), the characteristics shall be the same as those specified by the following interface:

SUBROUTINE my_read_routine_unformatted &
(dtv, &
unit, &
iostat, iomsg)

! the derived-type variable
dtv-type-spec, INTENT(INOUT) :: dtv
INTEGER, INTENT(IN) :: unit

! the edit descriptor string
CHARACTER (LEN=__), INTENT(IN) :: iotype
INTEGER, INTENT(IN) :: v_list(:)
INTEGER, INTENT(OUT) :: iostat
CHARACTER (LEN=__), INTENT(INOUT) :: iomsg

END

6 If the defined-io-generic-spec is WRITE (FORMATTED), the characteristics shall be the same as those specified by the following interface:

SUBROUTINE my_write_routine_formatted &
(dtv, &
unit, &
iotype, v_list, &
iostat, iomsg)

! the derived-type value/variable
dtv-type-spec, INTENT(IN) :: dtv
INTEGER, INTENT(IN) :: unit

! the edit descriptor string
CHARACTER (LEN=__), INTENT(IN) :: iotype
INTEGER, INTENT(IN) :: v_list(:)
INTEGER, INTENT(OUT) :: iostat
CHARACTER (LEN=__), INTENT(INOUT) :: iomsg

END

7 If the defined-io-generic-spec is WRITE (UNFORMATTED), the characteristics shall be the same as those specified by the following interface:

SUBROUTINE my_write_routine_unformatted &
(dtv, &
unit, &
iostat, iomsg)

! the derived-type value/variable
dtv-type-spec, INTENT(IN) :: dtv

END

8 If the defined-io-generic-spec is WRITE (UNFORMATTED), the characteristics shall be the same as those specified by the following interface:
The actual specific procedure names (the my_. . ._routine_. . . procedure names above) are not significant. In the discussion here and elsewhere, the dummy arguments in these interfaces are referred to by the names given above; the names are, however, arbitrary.

### 12.6.4.8.3 Executing defined input/output data transfers

1. If a defined input/output procedure is selected for an effective item as specified in 12.6.4.8.4, the processor shall call the selected defined input/output procedure for that item. The defined input/output procedure controls the actual data transfer operations for the derived-type list item.

2. A data transfer statement that includes a derived-type list item and that causes a defined input/output procedure to be invoked is called a parent data transfer statement. A data transfer statement that is executed while a parent data transfer statement is being processed and that specifies the unit passed into a defined input/output procedure is called a child data transfer statement.

#### NOTE 12.38

A defined input/output procedure will usually contain child data transfer statements that read values from or write values to the current record or at the current file position. The effect of executing the defined input/output procedure is similar to that of substituting the list items from any child data transfer statements into the parent data transfer statement’s list items, along with similar substitutions in the format specification.

#### NOTE 12.39

A particular execution of a READ, WRITE or PRINT statement can be both a parent and a child data transfer statement. A defined input/output procedure can indirectly call itself or another defined input/output procedure by executing a child data transfer statement containing a list item of derived type, where a matching interface is accessible for that derived type. If a defined input/output procedure calls itself indirectly in this manner, it cannot be declared NON_RECURSIVE.

3. A child data transfer statement is processed differently from a nonchild data transfer statement in the following ways.
   - Executing a child data transfer statement does not position the file prior to data transfer.
   - An unformatted child data transfer statement does not position the file after data transfer is complete.
   - Any ADVANCE= specifier in a child input/output statement is ignored.

4. When a defined input/output procedure is invoked, the processor shall pass a unit argument that has a value as follows.
   - If the parent data transfer statement uses a file-unit-number, the value of the unit argument shall be that of the file-unit-number.
   - If the parent data transfer statement is a WRITE statement with an asterisk unit or a PRINT statement, the unit argument shall have the same value as the named constant OUTPUT_UNIT of the intrinsic module ISO_FORTRAN_ENV (16.10.2).
   - If the parent data transfer statement is a READ statement with an asterisk unit or a READ statement without an io-control-spec-list, the unit argument shall have the same value as the INPUT_UNIT named constant of the intrinsic module ISO_FORTRAN_ENV (16.10.2).
   - Otherwise the parent data transfer statement accesses an internal file, in which case the unit argument shall have a processor-dependent negative value.
NOTE 12.40
The unit argument passed to a defined input/output procedure will be negative when the parent data transfer statement specified an internal unit, or specified an external unit that is a NEWUNIT value. When an internal unit is used with the INQUIRE statement, an error condition will occur, and any variable specified in an IOSTAT= specifier will be assigned the value IOSTAT_INQUIRE_INTERNAL_UNIT from the intrinsic module ISO_FORTRAN_ENV (16.10.2).

For formatted data transfer, the processor shall pass an iotype argument that has the value

• “LISTDIRECTED” if the parent data transfer statement specified list directed formatting,

• “NAMELIST” if the parent data transfer statement specified namelist formatting, or

• “DT” concatenated with the char-literal-constant, if any, of the DT edit descriptor in the format specification

of the parent data transfer statement.

If the parent data transfer statement is an input statement, the dtv dummy argument is argument associated with the effective item that caused the defined input procedure to be invoked, as if the effective item were an actual argument in this procedure reference (5.4.5).

If the parent data transfer statement is an output statement, the processor shall provide the value of the effective item in the dtv dummy argument.

If the v-list of the edit descriptor appears in the parent data transfer statement, the processor shall provide the values from it in the v_list dummy argument, with the same number of elements in the same order as v-list.

If there is no v-list in the edit descriptor or if the data transfer statement specifies list-directed or namelist formatting, the processor shall provide v_list as a zero-sized array.

NOTE 12.41
The user’s procedure might choose to interpret an element of the v_list argument as a field width, but this is not required. If it does, it would be appropriate to fill an output field with “*”s if the width is too small.

The iostat argument is used to report whether an error, end-of-record, or end-of-file condition (12.11) occurs. If an error condition occurs, the defined input/output procedure shall assign a positive value to the iostat argument. Otherwise, if an end-of-file condition occurs, the defined input procedure shall assign the value of the named constant IOSTAT_END (16.10.2.16) to the iostat argument. Otherwise, if an end-of-record condition occurs, the defined input procedure shall assign the value of the named constant IOSTAT_EOR (16.10.2.17) to iostat. Otherwise, the defined input/output procedure shall assign the value zero to the iostat argument.

If the defined input/output procedure returns a nonzero value for the iostat argument, the procedure shall also return an explanatory message in the iomsg argument. Otherwise, the procedure shall not change the value of the iomsg argument.

NOTE 12.42
The values of the iostat and iomsg arguments set in a defined input/output procedure need not be passed to all of the parent data transfer statements.

If the iostat argument of the defined input/output procedure has a nonzero value when that procedure returns, and the processor therefore terminates execution of the program as described in 12.11, the processor shall make the value of the iomsg argument available in a processor-dependent manner.

While a parent READ statement is active, an input/output statement shall not read from any external unit other than the one specified by the unit dummy argument and shall not perform output to any external unit.

While a parent WRITE or PRINT statement is active, an input/output statement shall not perform output to any external unit other than the one specified by the unit dummy argument and shall not read from any external unit.
While a parent data transfer statement is active, a data transfer statement that specifies an internal file is permitted.

OPEN, CLOSE, BACKSPACE, ENDFILE, and REWIND statements shall not be executed while a parent data transfer statement is active.

A defined input/output procedure may use a format specification with a DT edit descriptor for handling a component of the derived type that is itself of a derived type. A child data transfer statement that is a list directed or namelist input/output statement may contain a list item of derived type.

Because a child data transfer statement does not position the file prior to data transfer, the child data transfer statement starts transferring data from where the file was positioned by the parent data transfer statement’s most recently processed effective item or edit descriptor. This is not necessarily at the beginning of a record.

The edit descriptors T and TL used on unit by a child data transfer statement shall not cause the file to be positioned before the file position at the time the defined input/output procedure was invoked.

A defined input/output procedure could use INQUIRE to determine the settings of BLANK=, PAD=, ROUND=, DECIMAL=, and DELIM= for an external unit. The INQUIRE statement provides values as specified in 12.10.

Neither a parent nor child data transfer statement shall be asynchronous.

A defined input/output procedure, and any procedures invoked therefrom, shall not define, nor cause to become undefined, any storage unit referenced by any input/output list item, the corresponding format, or any specifier in any active parent data transfer statement, except through the dtv argument.

A child data transfer statement cannot specify the ID=, POS=, or REC= specifiers in an input/output control list.

A simple example of derived type formatted output follows. The derived type variable chairman has two components. The type and an associated write formatted procedure are defined in a module so as to be accessible from wherever they might be needed. It would also be possible to check that iotype indeed has the value ’DT’ and to set iostat and iomsg accordingly.

```
MODULE p

TYPE :: person
  CHARACTER (LEN=20) :: name
  INTEGER :: age
END TYPE person

CONTAINS
  SUBROUTINE pwf (dtv,unit,iotype,vlist,iostat,iomsg)
    ! argument definitions
    CLASS(person), INTENT(IN) :: dtv
    INTEGER, INTENT(IN) :: unit
    CHARACTER (LEN=*) , INTENT(IN) :: iotype
    CONTAINS
```

NOTE 12.43

NOTE 12.44

NOTE 12.45
NOTE 12.45 (cont.)

```
INTEGER, INTENT(IN) :: vlist(:)
INTEGER, INTENT(OUT) :: iostat
CHARACTER (LEN=*)) :: iomsg
!
local variable
CHARACTER (LEN=9) :: pfmt
!
! vlist(1) and (2) are to be used as the field widths of the two
! components of the derived type variable. First set up the format to
! be used for output.
WRITE(pfmt,'(A,I2,A,I2,A)' ) '(A', vlist(1), ',I', vlist(2), ')
!
! now the basic output statement
WRITE(unit, FMT=pfmt, IOSTAT=iostat) dtv%name, dtv%age

END SUBROUTINE pwf
END MODULE p
```

PROGRAM committee
  USE p
  INTEGER id, members
  TYPE (person) :: chairman
  ...
  WRITE(6, FMT="(I2, DT (15,6), I5)" ) id, chairman, members
  ! this writes a record with four fields, with lengths 2, 15, 6, 5
  ! respectively
END PROGRAM

NOTE 12.46

In the following example, the variables of the derived type node form a linked list, with a single value at each node. The subroutine pwf is used to write the values in the list, one per line.

```
MODULE p

  TYPE node
    INTEGER :: value = 0
    TYPE (node), POINTER :: next_node => NULL ( )
  CONTAINS
    PROCEDURE,PRIVATE :: pwf
    GENERIC :: WRITE(FORMATTED) => pwf
  END TYPE node

  CONTAINS

    SUBROUTINE pwf (dtv,unit,iotype,vlist,iostat,iomsg)
      ! Write the chain of values, each on a separate line in I9 format.
      CLASS(node), INTENT(IN) :: dtv
      INTEGER, INTENT(IN) :: unit
      CHARACTER (LEN=*) :: iotype
      INTEGER, INTENT(IN) :: vlist(:)
      INTEGER, INTENT(OUT) :: iostat
      CHARACTER (LEN=*) :: iomsg
      ...
```

NOTE 12.46 (cont.)

```fortran
WRITE(unit,'(i9 /)', IOSTAT = iostat) dtv%value
IF(iostat/=0) RETURN
IF(ASSOCIATED(dtv%next_node)) WRITE(unit,'(dt)', IOSTAT=iostat) dtv%next_node
END SUBROUTINE pwf
END MODULE p
```

12.6.4.8.4 Resolving defined input/output procedure references

1 A suitable generic interface for defined input/output of an effective item is one that has a defined-io-generic-spec that is appropriate to the direction (read or write) and form (formatted or unformatted) of the data transfer as specified in 12.6.4.8.2, and has a specific interface whose dtv argument is compatible with the effective item according to the rules for argument association in 15.5.2.4.

2 When an effective item (12.6.3) that is of derived type is encountered during a data transfer, defined input/output occurs if both of the following conditions are true.

   (1) The circumstances of the input/output are such that defined input/output is permitted; that is, either
   
   (a) the transfer was initiated by a list-directed, namelist, or unformatted input/output statement, or
   
   (b) a format specification is supplied for the data transfer statement, and the edit descriptor corresponding to the effective item is a DT edit descriptor.

   (2) A suitable defined input/output procedure is available; that is, either
   
   (a) the declared type of the effective item has a suitable generic type-bound procedure, or
   
   (b) a suitable generic interface is accessible.

3 If (2a) is true, the procedure referenced is determined as for explicit type-bound procedure references (15.5); that is, the binding with the appropriate specific interface is located in the declared type of the effective item, and the corresponding binding in the dynamic type of the effective item is selected.

4 If (2a) is false and (2b) is true, the reference is to the procedure identified by the appropriate specific interface in the interface block.

12.6.5 Termination of data transfer statements

1 Termination of a data transfer statement occurs when

- format processing encounters a colon or data edit descriptor and there are no remaining elements in the input-item-list or output-item-list,
- unformatted or list-directed data transfer exhausts the input-item-list or output-item-list,
- namelist output exhausts the namelist-group-object-list,
- an error condition occurs,
- an end-of-file condition occurs,
- a slash (/) is encountered as a value separator (13.10, 13.11) in the record being read during list-directed or namelist input, or
- an end-of-record condition occurs during execution of a nonadvancing input statement (12.11).
12.7 Waiting on pending data transfer

12.7.1 Wait operation

Execution of an asynchronous data transfer statement in which neither an error, end-of-record, nor end-of-file condition occurs initiates a pending data transfer operation. There may be multiple pending data transfer operations for the same or multiple units simultaneously. A pending data transfer operation remains pending until a corresponding wait operation is performed. A wait operation may be performed by a BACKSPACE, CLOSE, ENDFILE, FLUSH, INQUIRE, PRINT, READ, REWIND, WAIT, or WRITE statement.

A wait operation completes the processing of a pending data transfer operation. Each wait operation completes only a single data transfer operation, although a single statement may perform multiple wait operations.

If the actual data transfer is not yet complete, the wait operation first waits for its completion. If the data transfer operation is an input operation that completed without error, the storage units of the input/output storage sequence then become defined with the values as described in 12.6.2.15 and 12.6.4.5.

If any error, end-of-file, or end-of-record conditions occur, the applicable actions specified by the IOSTAT=, IOMSG=, ERR=, END=, and EOR= specifiers of the statement that performs the wait operation are taken.

If an error or end-of-file condition occurs during a wait operation for a unit, the processor performs a wait operation for all pending data transfer operations for that unit.

NOTE 12.47

Error, end-of-file, and end-of-record conditions can be raised either during the data transfer statement that initiates asynchronous input/output, a subsequent asynchronous data transfer statement for the same unit, or during the wait operation. If such conditions are raised during a data transfer statement, they trigger actions according to the IOSTAT=, ERR=, END=, and EOR= specifiers of that statement; if they are raised during the wait operation, the actions are in accordance with the specifiers of the statement that performs the wait operation.

After completion of the wait operation, the data transfer operation and its input/output storage sequence are no longer considered to be pending.

12.7.2 WAIT statement

A WAIT statement performs a wait operation for specified pending asynchronous data transfer operations.

R1222 wait-stmt  is  WAIT (wait-spec-list)

R1223 wait-spec  is  [UNIT = ] file-unit-number
    or  END = label
    or  EOR = label
    or  ERR = label
    or  ID = scalar-int-expr
    or  IOMSG = iomsg-variable
    or  IOSTAT = scalar-int-variable

C1236 No specifier shall appear more than once in a given wait-spec-list.

C1237 A file-unit-number shall be specified in a wait-spec-list; if the optional characters UNIT= are omitted, the file-unit-number shall be the first item in the wait-spec-list.

C1238 (R1223) The label in the ERR=, EOR=, or END= specifier shall be the statement label of a branch target statement that appears in the same inclusive scope as the WAIT statement.

The IOSTAT=, ERR=, EOR=, END=, and IOMSG= specifiers are described in 12.11.
The value of the expression specified in the ID= specifier shall be zero or the identifier of a pending data transfer operation for the specified unit. If the ID= specifier appears, a wait operation for the specified data transfer operation, if any, is performed. If the ID= specifier is omitted, wait operations for all pending data transfers for the specified unit are performed.

Execution of a WAIT statement specifying a unit that does not exist, has no file connected to it, or is not open for asynchronous input/output is permitted, provided that the WAIT statement has no ID= specifier; such a WAIT statement does not cause an error or end-of-file condition to occur.

**NOTE 12.48**

An EOR= specifier has no effect if the pending data transfer operation is not a nonadvancing read. An END= specifier has no effect if the pending data transfer operation is not a READ.

### 12.8 File positioning statements

#### 12.8.1 Syntax

1. **R1224** `backspace-stmt` is `BACKSPACE file-unit-number` or `BACKSPACE ( position-spec-list )`
2. **R1225** `endfile-stmt` is `ENDFILE file-unit-number` or `ENDFILE ( position-spec-list )`
3. **R1226** `rewind-stmt` is `REWIND file-unit-number` or `REWIND ( position-spec-list )`

1. A unit that is connected for direct access shall not be referred to by a BACKSPACE, ENDFILE, or REWIND statement. A unit that is connected for unformatted stream access shall not be referred to by a BACKSPACE statement. A unit that is connected with an ACTION= specifier having the value READ shall not be referred to by an ENDFILE statement.

2. **R1227** `position-spec` is `[ UNIT = ] file-unit-number` or `IOMSG = iomsg-variable` or `IOSTAT = scalar-int-variable` or `ERR = label`

3. No specifier shall appear more than once in a given `position-spec-list`.

4. A file-unit-number shall be specified in a `position-spec-list`; if the optional characters UNIT= are omitted, the file-unit-number shall be the first item in the `position-spec-list`.

5. (R1227) The label in the ERR= specifier shall be the statement label of a branch target statement that appears in the same inclusive scope as the file positioning statement.

6. The IOSTAT=, ERR=, and IOMSG= specifiers are described in 12.11.

7. Execution of a file positioning statement performs a wait operation for all pending asynchronous data transfer operations for the specified unit.

#### 12.8.2 BACKSPACE statement

1. Execution of a BACKSPACE statement causes the file connected to the specified unit to be positioned before the current record if there is a current record, or before the preceding record if there is no current record. If the file is at its initial point, the position of the file is not changed.
NOTE 12.49

If the preceding record is an endfile record, the file is positioned before the endfile record.

1. If a BACKSPACE statement causes the implicit writing of an endfile record, the file is positioned before the record that precedes the endfile record.
2. Backspacing a file that is connected but does not exist is prohibited.
3. Backspacing over records written using list-directed or namelist formatting is prohibited.

NOTE 12.50

An example of a BACKSPACE statement is:

BACKSPACE (10, IOSTAT = N)

12.8.3 ENDFILE statement

1. Execution of an ENDFILE statement for a file connected for sequential access writes an endfile record as the next record of the file. The file is then positioned after the endfile record, which becomes the last record of the file. If the file also may be connected for direct access, only those records before the endfile record are considered to have been written. Thus, only those records may be read during subsequent direct access connections to the file.
2. After execution of an ENDFILE statement for a file connected for sequential access, a BACKSPACE or REWIND statement shall be used to reposition the file prior to execution of any data transfer input/output statement or ENDFILE statement.
3. Execution of an ENDFILE statement for a file connected for stream access causes the terminal point of the file to become equal to the current file position. Only file storage units before the current position are considered to have been written; thus only those file storage units may be subsequently read. Subsequent stream output statements may be used to write further data to the file.
4. Execution of an ENDFILE statement for a file that is connected but does not exist creates the file; if the file is connected for sequential access, it is created prior to writing the endfile record.

NOTE 12.51

An example of an ENDFILE statement is:

ENDFILE K

12.8.4 REWIND statement

1. Execution of a REWIND statement causes the specified file to be positioned at its initial point.

NOTE 12.52

If the file is already positioned at its initial point, execution of this statement has no effect on the position of the file.

2. Execution of a REWIND statement for a file that is connected but does not exist is permitted and has no effect on any file.

NOTE 12.53

An example of a REWIND statement is:

REWIND 10
12.9 FLUSH statement

R1228 flush-stmt is FLUSH file-unit-number
or FLUSH ( flush-spec-list )

R1229 flush-spec is [UNIT =] file-unit-number
or IOSTAT = scalar-int-variable
or IOMSG = iomsg-variable
or ERR = label

C1242 No specifier shall appear more than once in a given flush-spec-list.

C1243 A file-unit-number shall be specified in a flush-spec-list; if the optional characters UNIT= are omitted from the unit specifier, the file-unit-number shall be the first item in the flush-spec-list.

C1244 (R1229) The label in the ERR= specifier shall be the statement label of a branch target statement that appears in the same inclusive scope as the FLUSH statement.

The IOSTAT=, IOMSG= and ERR= specifiers are described in 12.11.

Execution of a FLUSH statement causes data written to an external file to be available to other processes, or causes data placed in an external file by means other than Fortran to be available to a READ statement. These actions are processor dependent.

Execution of a FLUSH statement for a file that is connected but does not exist is permitted and has no effect on any file. A FLUSH statement has no effect on file position.

Execution of a FLUSH statement performs a wait operation for all pending asynchronous data transfer operations for the specified unit.

NOTE 12.54
Because this document does not specify the mechanism of file storage, the exact meaning of the flush operation is not precisely defined. It is expected that the flush operation will make all data written to a file available to other processes or devices, or make data recently added to a file by other processes or devices available to the program via a subsequent read operation. This is commonly called “flushing input/output buffers”.

NOTE 12.55
An example of a FLUSH statement is:

FLUSH (10, IOSTAT = N)

12.10 File inquiry statement

12.10.1 Forms of the INQUIRE statement

The INQUIRE statement can be used to inquire about properties of a particular named file, of the connection to a particular unit, or the number of file storage units required for an output list. There are three forms of the INQUIRE statement: inquire by file, which uses the FILE= specifier, inquire by unit, which uses the UNIT= specifier, and inquire by output list, which uses only the IOLength= specifier. Assignments to specifier variables are performed as if by intrinsic assignment.

For inquiry by unit, the unit specified need not exist or be connected to a file. If it is connected to a file, the inquiry is being made about the connection and about the file connected.

For inquiry by file, the file specified need not exist or be connected to a unit. If it is connected to a unit, the inquiry is being made about the connection as well as about the file.
An INQUIRE statement may be executed before, while, or after a file is connected to a unit. All values assigned by an INQUIRE statement are those that are current at the time the statement is executed.

```plaintext
4
R1230 inquire-stmt is INQUIRE ( inquire-spec-list )
or INQUIRE ( IOLENGTH = scalar-int-variable )
    output-item-list
```

**NOTE 12.56**

Examples of INQUIRE statements are:

```
INQUIRE (IOLENGTH = IOL) A (1:N)
INQUIRE (UNIT = JOAN, OPENED = LOG_01, NAMED = LOG_02, &
    FORM = CHAR_VAR, IOSTAT = IOS)
```

### 12.10.2 Inquiry specifiers

#### 12.10.2.1 Syntax

1. Unless constrained, the following inquiry specifiers may be used in either of the inquire by file or inquire by unit forms of the INQUIRE statement.

```plaintext
R1231 inquire-spec is [ UNIT = ] file-unit-number
or FILE = file-name-expr
or ACCESS = scalar-default-char-variable
or ACTION = scalar-default-char-variable
or ASYNCHRONOUS = scalar-default-char-variable
or BLANK = scalar-default-char-variable
or DECIMAL = scalar-default-char-variable
or DElim = scalar-default-char-variable
or DIRECT = scalar-default-char-variable
or ENCODING = scalar-default-char-variable
or ERR = label
or EXIST = scalar-logical-variable
or FORM = scalar-default-char-variable
or FORMATTED = scalar-default-char-variable
or ID = scalar-int-expr
or IOMSG = iomsg-variable
or IOSTAT = scalar-int-variable
or NAME = scalar-default-char-variable
or NAMED = scalar-logical-variable
or NEXTREC = scalar-int-variable
or NUMBER = scalar-int-variable
or OPENED = scalar-logical-variable
or PAD = scalar-default-char-variable
or PENDING = scalar-logical-variable
or POS = scalar-int-variable
or POSITION = scalar-default-char-variable
or READ = scalar-default-char-variable
or READWRITE = scalar-default-char-variable
or RECL = scalar-int-variable
or ROUND = scalar-default-char-variable
or SEQUENTIAL = scalar-default-char-variable
or SIGN = scalar-default-char-variable
or SIZE = scalar-int-variable
or STREAM = scalar-default-char-variable
or UNFORMATTED = scalar-default-char-variable
```
or  WRITE = scalar-default-char-variable

C1245  No specifier shall appear more than once in a given inquire-spec-list.

C1246  An inquire-spec-list shall contain one FILE= specifier or one file-unit-number, but not both.

C1247  In the inquire by unit form of the INQUIRE statement, if the optional characters UNIT= are omitted, the file-unit-number shall be the first item in the inquire-spec-list.

C1248  If an ID= specifier appears in an inquire-spec-list, a PENDING= specifier shall also appear.

C1249  (R1229) The label in the ERR= specifier shall be the statement label of a branch target statement that appears in the same inclusive scope as the INQUIRE statement.

2 If file-unit-number identifies an internal unit (12.6.4.8.2), an error condition occurs.

3 When a returned value of a specifier other than the NAME= specifier is of type character, the value returned is in upper case.

4 If an error condition occurs during execution of an INQUIRE statement, all of the inquiry specifier variables become undefined, except for variables in the IOSTAT= and IOMSG= specifiers (if any).

5 The IOSTAT=, ERR=, and IOMSG= specifiers are described in 12.11.

12.10.2.2  FILE= specifier in the INQUIRE statement

1 The value of the file-name-expr in the FILE= specifier specifies the name of the file being inquired about. The named file need not exist or be connected to a unit. The value of the file-name-expr shall be of a form acceptable to the processor as a file name. Any trailing blanks are ignored. The interpretation of case is processor dependent.

12.10.2.3  ACCESS= specifier in the INQUIRE statement

1 The scalar-default-char-variable in the ACCESS= specifier is assigned the value SEQUENTIAL if the connection is for sequential access, DIRECT if the connection is for direct access, or STREAM if the connection is for stream access. If there is no connection, it is assigned the value UNDEFINED.

12.10.2.4  ACTION= specifier in the INQUIRE statement

1 The scalar-default-char-variable in the ACTION= specifier is assigned the value READ if the connection is for input only, WRITE if the connection is for output only, and READWRITE if the connection is for both input and output. If there is no connection, the scalar-default-char-variable is assigned the value UNDEFINED.

12.10.2.5  ASYNCHRONOUS= specifier in the INQUIRE statement

1 The scalar-default-char-variable in the ASYNCHRONOUS= specifier is assigned the value YES if the connection allows asynchronous input/output; it is assigned the value NO if the connection does not allow asynchronous input/output. If there is no connection, the scalar-default-char-variable is assigned the value UNDEFINED.

12.10.2.6  BLANK= specifier in the INQUIRE statement

1 The scalar-default-char-variable in the BLANK= specifier is assigned the value ZERO or NULL, corresponding to the blank interpretation mode in effect for a connection for formatted input/output. If there is no connection, or if the connection is not for formatted input/output, the scalar-default-char-variable is assigned the value UNDEFINED.

12.10.2.7  DECIMAL= specifier in the INQUIRE statement

1 The scalar-default-char-variable in the DECIMAL= specifier is assigned the value COMMA or POINT, corresponding to the decimal edit mode in effect for a connection for formatted input/output. If there is no connection,
or if the connection is not for formatted input/output, the scalar-default-char-variable is assigned the value UNDEFINED.

12.10.2.8 DELIM= specifier in the INQUIRE statement

1 The scalar-default-char-variable in the DELIM= specifier is assigned the value APOSTROPHE, QUOTE, or NONE, corresponding to the delimiter mode in effect for a connection for formatted input/output. If there is no connection or if the connection is not for formatted input/output, the scalar-default-char-variable is assigned the value UNDEFINED.

12.10.2.9 DIRECT= specifier in the INQUIRE statement

1 The scalar-default-char-variable in the DIRECT= specifier is assigned the value YES if DIRECT is included in the set of allowed access methods for the file, NO if DIRECT is not included in the set of allowed access methods for the file, and UNKNOWN if the processor is unable to determine whether DIRECT is included in the set of allowed access methods for the file or if the unit identified by file-unit-number is not connected to a file.

12.10.2.10 ENCODING= specifier in the INQUIRE statement

1 The scalar-default-char-variable in the ENCODING= specifier is assigned the value UTF-8 if the connection is for formatted input/output with an encoding form of UTF-8, and is assigned the value UNDEFINED if the connection is for unformatted input/output. If there is no connection, it is assigned the value UTF-8 if the processor is able to determine that the encoding form of the file is UTF-8; if the processor is unable to determine the encoding form of the file or if the unit identified by file-unit-number is not connected to a file, the variable is assigned the value UNKNOWN.

NOTE 12.57

The value assigned could be something other than UTF-8, UNDEFINED, or UNKNOWN if the processor supports other specific encoding forms (e.g. UTF-16BE).

12.10.2.11 EXIST= specifier in the INQUIRE statement

1 Execution of an INQUIRE by file statement causes the scalar-logical-variable in the EXIST= specifier to be assigned the value true if there exists a file with the specified name; otherwise, false is assigned. Execution of an INQUIRE by unit statement causes true to be assigned if the specified unit exists; otherwise, false is assigned.

12.10.2.12 FORM= specifier in the INQUIRE statement

1 The scalar-default-char-variable in the FORM= specifier is assigned the value FORMATTED if the connection is for formatted input/output, and is assigned the value UNFORMATTED if the connection is for unformatted input/output. If there is no connection, it is assigned the value UNDEFINED.

12.10.2.13 FORMATTED= specifier in the INQUIRE statement

1 The scalar-default-char-variable in the FORMATTED= specifier is assigned the value YES if FORMATTED is included in the set of allowed forms for the file, NO if FORMATTED is not included in the set of allowed forms for the file, and UNKNOWN if the processor is unable to determine whether FORMATTED is included in the set of allowed forms for the file or if the unit identified by file-unit-number is not connected to a file.

12.10.2.14 ID= specifier in the INQUIRE statement

1 The value of the expression specified in the ID= specifier shall be the identifier of a pending data transfer operation for the specified unit. This specifier interacts with the PENDING= specifier (12.10.2.21).
12.10.2.15 NAME= specifier in the INQUIRE statement

1 The scalar-default-char-variable in the NAME= specifier is assigned the value of the name of the file if the file has a name; otherwise, it becomes undefined. The value assigned shall be suitable for use as the value of the file-name-expr in the FILE= specifier in an OPEN statement.

NOTE 12.58
If this specifier appears in an INQUIRE by file statement, its value is not necessarily the same as the name given in the FILE= specifier.

The processor could assign a file name qualified by a user identification, device, directory, or other relevant information.

2 The case of the characters assigned to scalar-default-char-variable is processor dependent.

12.10.2.16 NAMED= specifier in the INQUIRE statement

1 The scalar-logical-variable in the NAMED= specifier is assigned the value true if the file has a name; otherwise, it is assigned the value false.

12.10.2.17 NEXTREC= specifier in the INQUIRE statement

1 The scalar-int-variable in the NEXTREC= specifier is assigned the value \( n + 1 \), where \( n \) is the record number of the last record read from or written to the connection for direct access. If there is a connection but no records have been read or written since the connection, the scalar-int-variable is assigned the value 1. If there is no connection, the connection is not for direct access, or the position is indeterminate because of a previous error condition, the scalar-int-variable becomes undefined. If there are pending data transfer operations for the specified unit, the value assigned is computed as if all the pending data transfers had already completed.

12.10.2.18 NUMBER= specifier in the INQUIRE statement

1 Execution of an INQUIRE by file statement causes the scalar-int-variable in the NUMBER= specifier to be assigned the value of the external unit number of the unit that is connected to the file. If more than one unit on an image is connected to the file, which of the connected external unit numbers is assigned to the scalar-int-variable is processor dependent. If there is no unit connected to the file, the value −1 is assigned. Execution of an INQUIRE by unit statement causes the scalar-int-variable to be assigned the value of file-unit-number.

12.10.2.19 OPENED= specifier in the INQUIRE statement

1 Execution of an INQUIRE by file statement causes the scalar-logical-variable in the OPENED= specifier to be assigned the value true if the file specified is connected to a unit; otherwise, false is assigned. Execution of an INQUIRE by unit statement causes the scalar-logical-variable to be assigned the value true if the specified unit is connected to a file; otherwise, false is assigned.

12.10.2.20 PAD= specifier in the INQUIRE statement

1 The scalar-default-char-variable in the PAD= specifier is assigned the value YES or NO, corresponding to the pad mode in effect for a connection for formatted input/output. If there is no connection or if the connection is not for formatted input/output, the scalar-default-char-variable is assigned the value UNDEFINED.

12.10.2.21 PENDING= specifier in the INQUIRE statement

1 The PENDING= specifier is used to determine whether previously pending asynchronous data transfers are complete. A data transfer operation is previously pending if it is pending at the beginning of execution of the INQUIRE statement.
If an ID= specifier appears and the specified data transfer operation is complete, then the variable specified in the PENDING= specifier is assigned the value false and the INQUIRE statement performs the wait operation for the specified data transfer.

If the ID= specifier is omitted and all previously pending data transfer operations for the specified unit are complete, then the variable specified in the PENDING= specifier is assigned the value false and the INQUIRE statement performs wait operations for all previously pending data transfers for the specified unit.

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

**NOTE 12.59**

The processor has considerable flexibility in defining when it considers a transfer to be complete. Any of the following approaches could be used:

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

### 12.10.2.22 POS= specifier in the INQUIRE statement

The scalar-int-variable in the POS= specifier is assigned the number of the file storage unit immediately following the current position of a file connected for stream access. If the file is positioned at its terminal position, the variable is assigned a value one greater than the number of the highest-numbered file storage unit in the file. If there are pending data transfer operations for the specified unit, the value assigned is computed as if all the pending data transfers had already completed. If there is no connection, the file is not connected for stream access, or if the position of the file is indeterminate because of previous error conditions, the variable becomes undefined.

### 12.10.2.23 POSITION= specifier in the INQUIRE statement

The scalar-default-char-variable in the POSITION= specifier is assigned the value REWIND if the connection was opened for positioning at its initial point, APPEND if the connection was opened for positioning before its endfile record or at its terminal point, and ASIS if the connection was opened without changing its position. If there is no connection or if the file is connected for direct access, the scalar-default-char-variable is assigned the value UNDEFINED. If the file has been repositioned since the connection, the scalar-default-char-variable is assigned a processor-dependent value, which shall not be REWIND unless the file is positioned at its initial point and shall not be APPEND unless the file is positioned so that its endfile record is the next record or at its terminal point if it has no endfile record.

### 12.10.2.24 READ= specifier in the INQUIRE statement

The scalar-default-char-variable in the READ= specifier is assigned the value YES if READ is included in the set of allowed actions for the file, NO if READ is not included in the set of allowed actions for the file, and UNKNOWN if the processor is unable to determine whether READ is included in the set of allowed actions for the file or if the unit identified by file-unit-number is not connected to a file.

### 12.10.2.25 READWRITE= specifier in the INQUIRE statement

The scalar-default-char-variable in the READWRITE= specifier is assigned the value YES if READWRITE is included in the set of allowed actions for the file, NO if READWRITE is not included in the set of allowed actions
for the file, and UNKNOWN if the processor is unable to determine whether READWRITE is included in the set of allowed actions for the file or if the unit identified by `file-unit-number` is not connected to a file.

### 12.10.2.26 RECL= specifier in the INQUIRE statement

1. The `scalar-int-variable` in the RECL= specifier is assigned the value of the record length of a connection for direct access, or the value of the maximum record length of a connection for sequential access. If the connection is for formatted input/output, the length is the number of characters for all records that contain only characters of default kind. If the connection is for unformatted input/output, the length is measured in `file storage units`. If there is no connection, the `scalar-int-variable` is assigned the value −1, and if the connection is for stream access, the `scalar-int-variable` is assigned the value −2.

### 12.10.2.27 ROUND= specifier in the INQUIRE statement

1. The `scalar-default-char-variable` in the ROUND= specifier is assigned the value UP, DOWN, ZERO, NEAREST, COMPATIBLE, or PROCESSOR_DEFINED, corresponding to the input/output rounding mode in effect for a connection for formatted input/output. If there is no connection or if the connection is not for formatted input/output, the `scalar-default-char-variable` is assigned the value UNDEFINED. The processor shall return the value PROCESSOR_DEFINED only if the behavior of the input/output rounding mode is different from that of the UP, DOWN, ZERO, NEAREST, and COMPATIBLE modes.

### 12.10.2.28 SEQUENTIAL= specifier in the INQUIRE statement

1. The `scalar-default-char-variable` in the SEQUENTIAL= specifier is assigned the value YES if SEQUENTIAL is included in the set of allowed access methods for the file, NO if SEQUENTIAL is not included in the set of allowed access methods for the file, and UNKNOWN if the processor is unable to determine whether SEQUENTIAL is included in the set of allowed access methods for the file or if the unit identified by `file-unit-number` is not connected to a file.

### 12.10.2.29 SIGN= specifier in the INQUIRE statement

1. The `scalar-default-char-variable` in the SIGN= specifier is assigned the value PLUS, SUPPRESS, or PROCESSOR_DEFINED, corresponding to the sign mode in effect for a connection for formatted input/output. If there is no connection, or if the connection is not for formatted input/output, the `scalar-default-char-variable` is assigned the value UNDEFINED.

### 12.10.2.30 SIZE= specifier in the INQUIRE statement

1. The `scalar-int-variable` in the SIZE= specifier is assigned the size of the file in `file storage units`. If the file size cannot be determined or if the unit identified by `file-unit-number` is not connected to a file, the variable is assigned the value −1.

2. For a file that may be connected for stream access, the file size is the number of the highest-numbered `file storage unit` in the file.

3. For a file that may be connected for sequential or direct access, the file size may be different from the number of `storage units` implied by the data in the records; the exact relationship is processor dependent.

4. If there are pending data transfer operations for the specified unit, the value assigned is computed as if all the pending data transfers had already completed.

### 12.10.2.31 STREAM= specifier in the INQUIRE statement

1. The `scalar-default-char-variable` in the STREAM= specifier is assigned the value YES if STREAM is included in the set of allowed access methods for the file, NO if STREAM is not included in the set of allowed access methods for the file, and UNKNOWN if the processor is unable to determine whether STREAM is included in the set of allowed access methods for the file or if the unit identified by `file-unit-number` is not connected to a file.
12.10.2.32 UNFORMATTED= specifier in the INQUIRE statement

1 The scalar-default-char-variable in the UNFORMATTED= specifier is assigned the value YES if UNFORMATTED is included in the set of allowed forms for the file, NO if UNFORMATTED is not included in the set of allowed forms for the file, and UNKNOWN if the processor is unable to determine whether UNFORMATTED is included in the set of allowed forms for the file or if the unit identified by file-unit-number is not connected to a file.

12.10.2.33 WRITE= specifier in the INQUIRE statement

1 The scalar-default-char-variable in the WRITE= specifier is assigned the value YES if WRITE is included in the set of allowed actions for the file, NO if WRITE is not included in the set of allowed actions for the file, and UNKNOWN if the processor is unable to determine whether WRITE is included in the set of allowed actions for the file or if the unit identified by file-unit-number is not connected to a file.

12.10.3 Inquire by output list

1 The scalar-int-variable in the IOLENGTH= specifier is assigned the processor-dependent number of file storage units that would be required to store the data of the output list in an unformatted file. The value shall be suitable as a RECL= specifier in an OPEN statement that connects a file for unformatted direct access if data will be read from or written to the file using data transfer statements with an input/output list that specifies transfer of a sequence of objects having the same types, type parameters, and extents, in the same order as the output list in the INQUIRE statement.

2 The output list in an INQUIRE statement shall not contain any derived-type list items that require a defined input/output procedure as described in subclause 12.6.3. If a derived-type list item appears in the output list, the value returned for the IOLENGTH= specifier assumes that no defined input/output procedure will be invoked.

12.11 Error, end-of-record, and end-of-file conditions

12.11.1 Occurrence of input/output conditions

1 The set of input/output error conditions is processor dependent. Except as otherwise specified, when an error condition occurs or is detected is processor dependent.

2 An end-of-record condition occurs when a nonadvancing input statement attempts to transfer data from a position beyond the end of the current record, unless the file is a stream file and the current record is at the end of the file (an end-of-file condition occurs instead).

3 An end-of-file condition occurs when

- an endfile record is encountered during the reading of a file connected for sequential access,
- an attempt is made to read a record beyond the end of an internal file, or
- an attempt is made to read beyond the end of a stream file.

4 An end-of-file condition may occur at the beginning of execution of an input statement. An end-of-file condition also may occur during execution of a formatted input statement when more than one record is required by the interaction of the input list and the format. An end-of-file condition also may occur during execution of a stream input statement.

12.11.2 Error conditions and the ERR= specifier

1 If an error condition occurs during execution of an input/output statement, the position of the file becomes indeterminate.
2 If an error condition occurs during execution of an input/output statement that contains neither an ERR= nor IOSTAT= specifier, error termination is initiated. If an error condition occurs during execution of an input/output statement that contains either an ERR= specifier or an IOSTAT= specifier then:

1. processing of the input/output list, if any, terminates;
2. if the statement is a data transfer statement or the error occurs during a wait operation, all do-variables in the statement that initiated the transfer become undefined;
3. if an IOSTAT= specifier appears, the scalar-int-variable in the IOSTAT= specifier becomes defined as specified in 12.11.5;
4. if an IOMSG= specifier appears, the iomsg-variable becomes defined as specified in 12.11.6;
5. if the statement is a READ statement and it contains a SIZE= specifier, the scalar-int-variable in the SIZE= specifier becomes defined as specified in 12.6.2.15;
6. if the statement is a READ statement or the error condition occurs in a wait operation for a transfer initiated by a READ statement, all input items or namelist group objects in the statement that initiated the transfer become undefined;
7. if an ERR= specifier appears, a branch to the statement labeled by the label in the ERR= specifier occurs.

12.11.3 End-of-file condition and the END= specifier

1 If an end-of-file condition occurs during execution of an input/output statement that contains neither an END= specifier nor an IOSTAT= specifier, error termination is initiated. If an end-of-file condition occurs during execution of an input/output statement that contains either an END= specifier or an IOSTAT= specifier, and an error condition does not occur then:

1. processing of the input list, if any, terminates;
2. if the statement is a data transfer statement or the end-of-file condition occurs during a wait operation, all do-variables in the statement that initiated the transfer become undefined;
3. if the statement is an input statement or the end-of-file condition occurs during a wait operation for a transfer initiated by an input statement, all input list items or namelist group objects in the statement that initiated the transfer become undefined;
4. if the file specified in the input statement is an external record file, it is positioned after the endfile record;
5. if an IOSTAT= specifier appears, the scalar-int-variable in the IOSTAT= specifier becomes defined as specified in 12.11.5;
6. if an IOMSG= specifier appears, the iomsg-variable becomes defined as specified in 12.11.6;
7. if an END= specifier appears, a branch to the statement labeled by the label in the END= specifier occurs.

12.11.4 End-of-record condition and the EOR= specifier

1 If an end-of-record condition occurs during execution of an input/output statement that contains neither an EOR= specifier nor an IOSTAT= specifier, error termination is initiated. If an end-of-record condition occurs during execution of an input/output statement that contains either an EOR= specifier or an IOSTAT= specifier, and an error condition does not occur then:

1. if the pad mode has the value
   (a) YES, the record is padded with blanks to satisfy the effective item (12.6.4.5.3) and corresponding data edit descriptors that require more characters than the record contains,
   (b) NO, the input list item becomes undefined;
2. processing of the input list, if any, terminates;
3. if the statement is a data transfer statement or the end-of-record condition occurs during a wait operation, all do-variables in the statement that initiated the transfer become undefined;
4. the file specified in the input statement is positioned after the current record;
(5) if an IOSTAT= specifier appears, the scalar-int-variable in the IOSTAT= specifier becomes defined as specified in 12.11.5;

(6) if an IOMSG= specifier appears, the iomsg-variable becomes defined as specified in 12.11.6;

(7) if a SIZE= specifier appears, the scalar-int-variable in the SIZE= specifier becomes defined as specified in (12.6.2.15);

(8) if an EOR= specifier appears, a branch to the statement labeled by the label in the EOR= specifier occurs.

12.11.5 IOSTAT= specifier

Execution of an input/output statement containing the IOSTAT= specifier causes the scalar-int-variable in the IOSTAT= specifier to become defined with

- a zero value if neither an error condition, an end-of-file condition, nor an end-of-record condition occurs,
- the processor-dependent positive integer value of the constant IOSTAT_INQUIRE_INTERNAL_UNIT from the intrinsic module ISO_FORTRAN_ENV (16.10.2) if a unit number in an INQUIRE statement identifies an internal file,
- a processor-dependent positive integer value different from IOSTAT_INQUIRE_INTERNAL_UNIT if any other error condition occurs,
- the processor-dependent negative integer value of the constant IOSTAT_END (16.10.2.16) from the intrinsic module ISO_FORTRAN_ENV if an end-of-file condition occurs and no error condition occurs,
- the processor-dependent negative integer value of the constant IOSTAT_EOR (16.10.2.17) from the intrinsic module ISO_FORTRAN_ENV if an end-of-record condition occurs and no error condition or end-of-file condition occurs, or
- a processor-dependent negative integer value different from IOSTAT_EOR and IOSTAT_END, if the IOSTAT= specifier appears in a FLUSH statement and the processor does not support the flush operation for the specified unit.

NOTE 12.60
An end-of-file condition can occur only for sequential or stream input and an end-of-record condition can occur only for nonadvancing input.

For example,

```fortran
READ (FMT = "(E8.3)", UNIT = 3, IOSTAT = IOSS) X
IF (IOSS < 0) THEN
   ! Perform end-of-file processing on the file connected to unit 3.
   CALL END_PROCESSING
ELSE IF (IOSS > 0) THEN
   ! Perform error processing
   CALL ERROR_PROCESSING
END IF
```

12.11.6 IOMSG= specifier

If an error, end-of-file, or end-of-record condition occurs during execution of an input/output statement, iomsg-variable is assigned an explanatory message as if by intrinsic assignment. If no such condition occurs, the definition status and value of iomsg-variable are unchanged.

12.12 Restrictions on input/output statements

If a unit, or a file connected to a unit, does not have all of the properties required for the execution of certain input/output statements, those statements shall not refer to the unit.
An input/output statement that is executed while another input/output statement is being executed is a recursive input/output statement. A recursive input/output statement shall not identify an external unit that is identified by another input/output statement being executed except that a child data transfer statement may identify its parent data transfer statement external unit.

An input/output statement shall not cause the value of any established format specification to be modified.

A recursive input/output statement shall not modify the value of any internal unit except that a recursive WRITE statement may modify the internal unit identified by that recursive WRITE statement.

The value of a specifier in an input/output statement shall not depend on the definition or evaluation of any other specifier in the io-control-spec-list or inquire-spec-list in that statement. The value of an internal-file-variable or of a FMT=, ID=, IOMSG=, IOSTAT=, or SIZE= specifier shall not depend on the value of any input-item or io-implied-do do-variable in the same statement.

The value of any subscript or substring bound of a variable that appears in a specifier in an input/output statement shall not depend on any input-item, io-implied-do do-variable, or on the definition or evaluation of any other specifier in the io-control-spec-list or inquire-spec-list in that statement.

In a data transfer statement, the variable specified in an IOSTAT=, IOMSG=, or SIZE= specifier, if any, shall not be associated with any entity in the data transfer input/output list (12.6.3) or namelist-group-object-list, nor with a do-variable of an io-implied-do in the data transfer input/output list.

In a data transfer statement, if a variable specified in an IOSTAT=, IOMSG=, or SIZE= specifier is an array element reference, its subscript values shall 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-list.

A variable that may become defined or undefined as a result of its use in a specifier in an INQUIRE statement, or any associated entity, shall not appear in another specifier in the same INQUIRE statement.

NOTE 12.61

Restrictions on the evaluation of expressions (10.1.4) prohibit certain side effects.
13 Input/output editing

13.1 Format specifications

1 A format used in conjunction with a data transfer statement provides information that directs the editing between
the internal representation of data and the characters of a sequence of formatted records.

2 A format (12.6.2.2) in a data transfer statement may refer to a FORMAT statement or to a character expression
that contains a format specification. A format specification provides explicit editing information. The format
alternatively may be an asterisk (*), which indicates list-directed formatting (13.10). Namelist formatting (13.11)
may be indicated by specifying a namelist-group-name instead of a format.

13.2 Explicit format specification methods

13.2.1 FORMAT statement

R1301 format-stmt is FORMAT format-specification
R1302 format-specification is ([ format-items ])
or ([ format-items, unlimited-format-item ])
C1301 (R1301) The format-stmt shall be labeled.

Blank characters may precede the initial left parenthesis of the format specification. Additional blank characters
may appear at any point within the format specification, with no effect on the interpretation of the format
specification, except within a character string edit descriptor (13.9).

NOTE 13.1
Examples of FORMAT statements are:

5 FORMAT (1PE12.4, I10)
9 FORMAT (I12, /, ' Dates: ', 2 (2I3, I5))

13.2.2 Character format specification

1 A character expression used as a format in a formatted input/output statement shall evaluate to a character
string whose leading part is a valid format specification.

NOTE 13.2
The format specification begins with a left parenthesis and ends with a right parenthesis.

2 All character positions up to and including the final right parenthesis of the format specification shall be defined
at the time the data transfer statement is executed, and shall not become redefined or undefined during the
execution of the statement. Character positions, if any, following the right parenthesis that ends the format
specification need not be defined and may contain any character data with no effect on the interpretation of the
format specification.

3 If the format is a character array, it is treated as if all of the elements of the array were specified in array element
order and were concatenated. However, if a format is a character array element, the format specification shall be
entirely within that array element.
NOTE 13.3

If a character constant is used as a *format* in *data transfer statement*, care shall be taken that the value of the character constant is a valid format specification. In particular, if a format specification delimited by apostrophes contains a character constant edit descriptor delimited with apostrophes, two apostrophes shall be written to delimit the edit descriptor and four apostrophes shall be written for each apostrophe that occurs within the edit descriptor. For example, the text:

```
2 ISN’T 3
```

can be written by various combinations of output statements and format specifications:

```
WRITE (6, 100) 2, 3
100 FORMAT (1X, I1, 1X, ’ISN’’T’, 1X, I1)
   WRITE (6, ’(1X, I1, 1X, ’’ISN’’’’T’’’, 1X, I1)’) 2, 3
   WRITE (6, ’(A)’) ’2 ISN’’T’’3’
```

Doubling of internal apostrophes usually can be avoided by using quotation marks to delimit the format specification and doubling of internal quotation marks usually can be avoided by using apostrophes as delimiters.

13.3 Form of a format item list

13.3.1 Syntax

R1303  *format-items* is *format-item [ , ] format-item* ...

R1304  *format-item* is *[ r ] data-edit-desc or control-edit-desc or char-string-edit-desc or *[ r ] ( format-items )

R1305  *unlimited-format-item* is * * ( format-items )

R1306  *r* is int-literal-constant

C1302  (R1303) The optional comma shall not be omitted except

- between a P edit descriptor and an immediately following F, E, EN, ES, D, or G edit descriptor (13.8.5), possibly preceded by a repeat specification,
- before a slash edit descriptor when the optional repeat specification does not appear (13.8.2),
- after a slash edit descriptor, or
- before or after a colon edit descriptor (13.8.3)

C1303  (R1305) An *unlimited-format-item* shall contain at least one data edit descriptor.

C1304  (R1306) *r* shall be positive.

C1305  (R1306) A kind parameter shall not be specified for *r*.

1 The integer literal constant *r* is called a repeat specification.

13.3.2 Edit descriptors

An edit descriptor is a data edit descriptor (*data-edit-desc*), control edit descriptor (*control-edit-desc*), or character string edit descriptor (*char-string-edit-desc*).
R1307  data-edit-desc is I w [. m ]
or B w [. m ]
or O w [. m ]
or Z w [. m ]
or F w . d
or E w . d [ E e ]
or EN w . d [ E e ]
or ES w . d [ E e ]
or EX w . d [ E e ]
or G w [. d [ E e ]]
or L w
or A [ w ]
or D w . d
or DT [ char-literal-constant ] [ ( v-list ) ]

R1308  w is int-literal-constant
R1309  m is int-literal-constant
R1310  d is int-literal-constant
R1311  e is int-literal-constant
R1312  v is signed-int-literal-constant

C1306  (R1308)  w shall be zero or positive for the I, B, O, Z, D, E, EN, ES, EX, F, and G edit descriptors.  w shall be positive for all other edit descriptors.

C1307  (R1307)  For the G edit descriptor, d shall be specified if w is not zero.

C1308  (R1307)  For the G edit descriptor, e shall not be specified if w is zero.

C1309  (R1307)  A kind parameter shall not be specified for the char-literal-constant in the DT edit descriptor, or for w, m, d, e, and v.

An I, B, O, Z, F, E, EN, ES, EX, G, L, A, D, or DT edit descriptor indicates the manner of editing.

R1313  control-edit-desc is position-edit-desc
or [ r ] /
or :
or sign-edit-desc
or k P
or blank-interp-edit-desc
or round-edit-desc
or decimal-edit-desc

R1314  k is signed-int-literal-constant

C1310  (R1314)  A kind parameter shall not be specified for k.

In k P, k is called the scale factor.

R1315  position-edit-desc is T n
or TL n
or TR n
or n X

R1316  n is int-literal-constant

C1311  (R1316)  n shall be positive.
C1312 (R1316) A kind parameter shall not be specified for \( n \).

R1317 \textit{sign-edit-desc} is SS

or SP

or S

R1318 \textit{blank-interp-edit-desc} is BN

or BZ

R1319 \textit{round-edit-desc} is RU

or RD

or RZ

or RN

or RC

or RP

R1320 \textit{decimal-edit-desc} is DC

or DP

4 A T, TL, TR, X, slash, colon, SS, SP, S, P, BN, BZ, RU, RD, RZ, RN, RC, RP, DC, or DP edit descriptor
indicates the manner of editing.

R1321 \textit{char-string-edit-desc} is \textit{char-literal-constant}

C1313 (R1321) A kind parameter shall not be specified for the \textit{char-literal-constant}.

5 Each \textit{rep-char} in a character string edit descriptor shall be capable of representation by the processor.

6 A character string edit descriptor provides constant data to be output, and is not valid for input.

7 The edit descriptors are without regard to case except within a character string edit descriptor.

\section*{13.3.3 Fields}

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

\section*{13.4 Interaction between input/output list and format}

1 The start of formatted data transfer using a format specification initiates format control (12.6.4.5.3). Each action
of format control depends on information jointly provided by the next edit descriptor in the format specification
and the next \textit{effective item} in the input/output list, if one exists.

2 If an input/output list specifies at least one \textit{effective item}, at least one data edit descriptor shall exist in the
format specification.

\textbf{NOTE 13.4}

An empty format specification of the form ( ) can be used only if the input/output list has no \textit{effective item}
(12.6.4.5). A zero length character item is an \textit{effective item}, but a zero sized array and an implied DO list
with an iteration count of zero is not.

3 A format specification is interpreted from left to right. The exceptions are format items preceded by a repeat
specification \( r \), and format reversion (described below).

4 A format item preceded by a repeat specification is processed as a list of \( r \) items, each identical to the format
item but without the repeat specification and separated by commas.
NOTE 13.5
An omitted repeat specification is treated in the same way as a repeat specification whose value is one.

To each data edit descriptor interpreted in a format specification, there corresponds one effective item specified by the input/output list (12.6.3), except that an input/output list item of type complex requires the interpretation of two F, E, EN, ES, D, or G edit descriptors. For each control edit descriptor or character edit descriptor, there is no corresponding item specified by the input/output list, and format control communicates information directly with the record.

Whenever format control encounters a data edit descriptor in a format specification, it determines whether there is a corresponding effective item specified by the input/output list. If there is such an item, it transmits appropriately edited information between the item and the record, and then format control proceeds. If there is no such item, format control terminates.

If format control encounters a colon edit descriptor in a format specification and another effective item is not specified, format control terminates.

If format control encounters the rightmost parenthesis of an unlimited format item, control reverts to the leftmost parenthesis of that unlimited format item. This reversion of format control has no effect on the changeable modes (12.5.2).

If format control encounters the rightmost parenthesis of a complete format specification and another effective item is not specified, format control terminates. However, if another effective item is specified, format control then reverts to the beginning of the format item terminated by the last preceding right parenthesis that is not part of a DT edit descriptor. If there is no such preceding right parenthesis, format control reverts to the first left parenthesis of the format specification. If any reversion occurs, the reused portion of the format specification shall contain at least one data edit descriptor. If format control reverts to a parenthesis that is preceded by a repeat specification, the repeat specification is reused. Reversion of format control, of itself, has no effect on the changeable modes. The file is positioned in a manner identical to the way it is positioned when a slash edit descriptor is processed (13.8.2).

NOTE 13.6
Example: The format specification:

```plaintext
10 FORMAT (1X, 2(F10.3, I5))
```

with an output list of

```plaintext
WRITE (10,10) 10.1, 3, 4.7, 1, 12.4, 5, 5.2, 6
```

produces the same output as the format specification:

```plaintext
```

NOTE 13.7
The effect of an unlimited-format-item is as if its enclosed list were preceded by a very large repeat count. There is no file positioning implied by unlimited-format-item reversion. This can be used to write what is commonly called a comma separated value record.

For example,

```plaintext
WRITE ( 10, ’( "IARRAY =", *( I0, :, ",")’)) IARRAY
```

produces a single record with a header and a comma separated list of integer values.
13.5 Positioning by format control

After each data edit descriptor or character string edit descriptor is processed, the file is positioned after the last character read or written in the current record.

After each T, TL, TR, or X edit descriptor is processed, the file is positioned as described in 13.8.1.1. After each slash edit descriptor is processed, the file is positioned as described in 13.8.2.

During formatted stream output, processing of an A edit descriptor can cause file positioning to occur (13.7.4). If format control reverts as described in 13.4, the file is positioned in a manner identical to the way it is positioned when a slash edit descriptor is processed (13.8.2).

During a read operation, any unprocessed characters of the current record are skipped whenever the next record is read.

13.6 Decimal symbol

The decimal symbol is the character that separates the whole and fractional parts in the decimal representation of a real number in an internal or external file. When the decimal edit mode is POINT, the decimal symbol is a decimal point. When the decimal edit mode is COMMA, the decimal symbol is a comma.

If the decimal edit mode is COMMA during list-directed input/output, the character used as a value separator is a semicolon in place of a comma.

13.7 Data edit descriptors

13.7.1 Purpose of data edit descriptors

Data edit descriptors cause the conversion of data to or from its internal representation; during formatted stream output, the A data edit descriptor may also cause file positioning. On input, the specified variable becomes defined unless an error condition, an end-of-file condition, or an end-of-record condition occurs. On output, the specified expression is evaluated.

During input from a Unicode file,

- characters in the record that correspond to an ASCII character variable shall have a position in the ISO 10646 character collating sequence of 127 or less, and
- characters in the record that correspond to a default character variable shall be representable as default characters.

During input from a non-Unicode file,

- characters in the record that correspond to a character variable shall have the kind of the character variable, and
- characters in the record that correspond to a numeric or logical variable shall be default characters.

During output to a Unicode file, all characters transmitted to the record are of ISO 10646 character kind. If a character input/output list item or character string edit descriptor contains a character that is not representable as an ISO 10646 character, the result is processor dependent.

During output to a non-Unicode file, characters transmitted to the record as a result of processing a character string edit descriptor or as a result of evaluating a numeric, logical, or default character data entity, are of default kind.
13.7.2 Numeric editing

13.7.2.1 General rules

1 The I, B, O, Z, F, E, EN, ES, EX, D, and G edit descriptors may be used to specify the input/output of integer, real, and complex data. The following general rules apply.

(1) On input, leading blanks are not significant. When the input field is not an IEEE exceptional specification or hexadecimal-significand number (13.7.2.3.2), the interpretation of blanks, other than leading blanks, is determined by the blank interpretation mode (13.8.6). Plus signs may be omitted. A field containing only blanks is considered to be zero.

(2) On input, with F, E, EN, ES, EX, D, and G editing, a decimal symbol appearing in the input field overrides the portion of an edit descriptor that specifies the decimal symbol location. The input field may have more digits than the processor uses to approximate the value of the datum.

(3) On output with I, F, E, EN, ES, EX, D, and G editing, the representation of a positive or zero internal value in the field may be prefixed with a plus sign, as controlled by the S, SP, and SS edit descriptors or the processor. The representation of a negative internal value in the field shall be prefixed with a minus sign.

(4) On output, the representation is right justified in the field. If the number of characters produced by the editing is smaller than the field width, leading blanks are inserted in the field.

(5) On output, if an exponent exceeds its specified or implied width using the E, EN, ES, EX, D, or G edit descriptor, or the number of characters produced exceeds the field width, the processor shall fill the entire field of width \( w \) with asterisks. However, the processor shall not produce asterisks if the field width is not exceeded when optional characters are omitted.

NOTE 13.8

When the sign mode is PLUS, a plus sign is not optional.

(6) On output, with I, B, O, Z, D, E, EN, ES, EX, F, and G editing, the specified value of the field width \( w \) may be zero. In such cases, the processor selects the smallest positive actual field width that does not result in a field filled with asterisks. The specified value of \( w \) shall not be zero on input.

(7) On output of a real zero value, the digits in the exponent field shall all be zero.

13.7.2.2 Integer editing

1 The \( I^w \) and \( I^{w.m} \) edit descriptors indicate that the field to be edited occupies \( w \) positions, except when \( w \) is zero. When \( w \) is zero, the processor selects the field width. On input, \( w \) shall not be zero. The specified input/output list item shall be of type integer. The G, B, O, and Z edit descriptor also may be used to edit integer data (13.7.5.2.1, 13.7.2.4).

2 On input, \( m \) has no effect.

3 In the standard form of the input field for the I edit descriptor, the character string is a signed-digit-string (R710), except for the interpretation of blanks. If the input field does not have the standard form and is not acceptable to the processor, an error condition occurs.

4 The output field for the \( I^w \) edit descriptor consists of zero or more leading blanks followed by a minus sign if the internal value is negative, or an optional plus sign otherwise, followed by the magnitude of the internal value as a digit-string without leading zeros.

NOTE 13.9

A digit-string always consists of at least one digit.

5 The output field for the \( I^{w.m} \) edit descriptor is the same as for the \( I^w \) edit descriptor, except that the digit-string consists of at least \( m \) digits. If necessary, sufficient leading zeros are included to achieve the minimum of \( m \) digits. The value of \( m \) shall not exceed the value of \( w \), except when \( w \) is zero. If \( m \) is zero and the internal value is zero,
the output field consists of only blank characters, regardless of the sign control in effect. When \( m \) and \( w \) are both zero, and the internal value is zero, one blank character is produced.

### 13.7.2.3 Real and complex editing

#### 13.7.2.3.1 General

1. The F, E, EN, ES, and D edit descriptors specify the editing of real and complex data. An input/output list item corresponding to an F, E, EN, ES, or D edit descriptor shall be real or complex. The G, B, O, and Z edit descriptors also may be used to edit real and complex data (13.7.5.2.2, 13.7.2.4).

#### 13.7.2.3.2 F editing

1. The F \( w.d \) edit descriptor indicates that the field occupies \( w \) positions, except when \( w \) is zero in which case the processor selects the field width. The fractional part of the field consists of \( d \) digits. On input, \( w \) shall not be zero.

2. A lower-case letter is equivalent to the corresponding upper-case letter in an IEEE exceptional specification or the exponent in a numeric input field.

3. The standard form of the input field is an IEEE exceptional specification, a hexadecimal-significand number, or consists of a mantissa optionally followed by an exponent. The form of the mantissa is an optional sign, followed by a string of one or more digits optionally containing a decimal symbol, including any blanks interpreted as zeros. The \( d \) has no effect on input if the input field contains a decimal symbol. If the decimal symbol is omitted, the rightmost \( d \) digits of the string, with leading zeros assumed if necessary, are interpreted as the fractional part of the value represented. The string of digits may contain more digits than a processor uses to approximate the value. The form of the exponent is one of the following:

   - a sign followed by a digit-string;
   - the letter E followed by zero or more blanks, followed by a signed-digit-string;
   - the letter D followed by zero or more blanks, followed by a signed-digit-string.

4. An exponent containing a D is processed identically to an exponent containing an E.

#### NOTE 13.10

If the input field does not contain an exponent, the effect is as if the basic form were followed by an exponent with a value of \( -k \), where \( k \) is the established scale factor (13.8.5).

5. An input field that is an IEEE exceptional specification consists of optional blanks, followed by either

   - an optional sign, followed by the string 'INF' or the string 'INFINITY', or
   - an optional sign, followed by the string 'NAN', optionally followed by zero or more alphanumeric characters enclosed in parentheses,

   optionally followed by blanks.

6. The value specified by 'INF' or 'INFINITY' is an IEEE infinity; this form shall not be used if the processor does not support IEEE infinities for the input variable. The value specified by 'NAN' is an IEEE NaN; this form shall not be used if the processor does not support IEEE NaNs for the input variable. The NaN value is a quiet NaN if the only nonblank characters in the field are 'NAN' or 'NAN()'; otherwise, the NaN value is processor dependent. The interpretation of a sign in a NaN input field is processor dependent.

7. An input field that is a hexadecimal-significand number consists of an optional sign, followed by the hexadecimal indicator which is the digit 0 immediately followed by the letter X, followed by a hexadecimal significand followed by a hexadecimal exponent. A hexadecimal significand is a string of one or more hexadecimal characters optionally containing a decimal symbol. The decimal symbol indicates the position of the hexadecimal point; if no decimal symbol appears, the hexadecimal point implicitly follows the last hexadecimal symbol. A hexadecimal exponent
is the letter P followed by a (decimal) signed-digit-string. Embedded blanks are not permitted in a hexadecimal-significand number. The value is equal to the significand multiplied by two raised to the power of the exponent, negated if the optional sign is minus.

8 If the input field does not have one of the standard forms, and is not acceptable to the processor, an error condition occurs.

9 For an internal value that is an IEEE infinity, the output field consists of blanks, if necessary, followed by a minus sign for negative infinity or an optional plus sign otherwise, followed by the letters 'Inf' or 'Infinity', right justified within the field. The minimum field width required for output of the form 'Inf' is 3 if no sign is produced, and 4 otherwise. The minimum field width required for output of the form 'Infinity' is 8 if no sign is produced, and 9 otherwise. If \( w \) is greater than or equal to the minimum required for the form 'Infinity', the form 'Infinity' is output. If \( w \) is zero or \( w \) is less than the minimum required for the form 'Infinity' and greater than or equal to the minimum required for the form 'Inf', the form 'Inf' is output. Otherwise (\( w \) is greater than zero but less than the minimum required for any form), the field is filled with asterisks.

For an internal value that is an IEEE NaN, the output field consists of blanks, if necessary, followed by the letters 'NaN' and optionally followed by one to \( w-5 \) alphanumeric processor-dependent characters enclosed in parentheses, right justified within the field. If \( w \) is greater than zero and less than 3, the field is filled with asterisks. If \( w \) is zero, the output field is 'NaN'.

NOTE 13.11

The processor-dependent characters following 'NaN' might convey additional information about that particular NaN.

11 For an internal value that is neither an IEEE infinity nor a NaN, the output field consists of blanks, if necessary, followed by a minus sign if the internal value is negative, or an optional plus sign otherwise, followed by a string of digits that contains a decimal symbol and represents the magnitude of the internal value, as modified by the established scale factor and rounded (13.7.2.3.8) to \( d \) fractional digits. Leading zeros are not permitted except for an optional zero immediately to the left of the decimal symbol if the magnitude of the value in the output field is less than one. The optional zero shall appear if there would otherwise be no digits in the output field.

13.7.2.3.3 E and D editing

1 The \( E.w.d \), \( D.w.d \), and \( E.w.d \) \( E \) edit descriptors indicate that the external field occupies \( w \) positions, except when \( w \) is zero in which case the processor selects the field width. The fractional part of the field contains \( d \) digits, unless a scale factor greater than one is in effect. If \( e \) is positive the exponent part contains \( e \) digits, otherwise it contains the minimum number of digits required to represent the exponent value. The \( e \) has no effect on input.

2 The form and interpretation of the input field is the same as for \( F.w.d \) editing (13.7.2.3.2).

3 For an internal value that is an IEEE infinity or NaN, the form of the output field is the same as for \( F.w.d \).

4 For an internal value that is neither an IEEE infinity nor a NaN, the form of the output field for a scale factor of zero is

\[
[\pm \cdot][0].x_1x_2\ldots x_d exp
\]

where:

- \( \pm \) signifies a plus sign or a minus sign;
- \( \cdot \) signifies a decimal symbol (13.6);
- \( x_1x_2\ldots x_d \) are the \( d \) most significant digits of the internal value after rounding (13.7.2.3.8);
- \( exp \) is a decimal exponent having one of the forms specified in table 13.1.
Table 13.1: E and D exponent forms

| Edit Descriptor | Absolute Value of Exponent | Form of Exponent
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>E(\cdot)d</td>
<td>(</td>
<td>\text{exp}</td>
</tr>
<tr>
<td>E(\cdot)d (\text{Ex with } e &gt; 0)</td>
<td>(</td>
<td>\text{exp}</td>
</tr>
<tr>
<td>E(\cdot)d (\text{E}0)</td>
<td>any</td>
<td>E±(z_1)(z_2) ... (z_e)</td>
</tr>
<tr>
<td>D(\cdot)d</td>
<td>(</td>
<td>\text{exp}</td>
</tr>
</tbody>
</table>

(1) where each \(z\) is a digit, and \(s\) is the minimum number of digits required to represent the exponent.

5 The sign in the exponent is produced. A plus sign is produced if the exponent value is zero.

6 The scale factor \(k\) controls the decimal normalization (13.3.2, 13.8.5). If \(-d < k \leq 0\), the output field contains exactly \(|k|\) leading zeros and \(d - |k|\) significant digits after the decimal symbol. If \(0 < k < d + 2\), the output field contains exactly \(k\) significant digits to the left of the decimal symbol and \(d - k + 1\) significant digits to the right of the decimal symbol. Other values of \(k\) are not permitted.

13.7.2.3.4 EN editing

1 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 three and the absolute value of the significand (R715) 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.

2 The forms of the edit descriptor are EN\(\cdot\)d and EN\(\cdot\)d \(\text{E}e\) indicating that the external field occupies \(w\) positions, except when \(w\) is zero in which case the processor selects the field width. The fractional part of the field contains \(d\) digits. If \(e\) is positive the exponent part contains \(e\) digits, otherwise it contains the minimum number of digits required to represent the exponent value.

3 The form and interpretation of the input field is the same as for F\(\cdot\)d editing (13.7.2.3.2).

4 For an internal value that is an IEEE infinity or NaN, the form of the output field is the same as for F\(\cdot\)d.

5 For an internal value that is neither an IEEE infinity nor a NaN, the form of the output field is

\[
[\pm] \text{yyyy} . \, x_1x_2\ldots x_d\text{exp}
\]

where:

\[\pm\] signifies a plus sign or a minus sign;

\(\text{yyyy}\) are the 1 to 3 decimal digits representative of the most significant digits of the internal value after rounding (13.7.2.3.8);

\(\text{yyyy}\) is an integer such that \(1 \leq \text{yyyy} < 1000\) or, if the output value is zero, \(\text{yyyy} = 0\);

\(\ldots\) signifies a decimal symbol (13.6);

\(x_1x_2\ldots x_d\) are the \(d\) next most significant digits of the internal value after rounding;

\(\text{exp}\) is a decimal exponent, divisible by three, having one of the forms specified in table 13.2.
Table 13.2: EN exponent forms

<table>
<thead>
<tr>
<th>Edit Descriptor</th>
<th>Absolute Value of Exponent</th>
<th>Form of Exponent¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN(w.d)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>\exp</td>
<td>\leq 99)</td>
</tr>
<tr>
<td>99 &lt; (</td>
<td>\exp</td>
<td>\leq 999)</td>
</tr>
<tr>
<td>EN(w.d) E(e) with (e &gt; 0)</td>
<td>(</td>
<td>\exp</td>
</tr>
<tr>
<td>EN(w.d) E0</td>
<td>any</td>
<td>E±(z_1z_2\ldots z_s)</td>
</tr>
</tbody>
</table>

ⁱ(1) where each \(z\) is a digit, and \(s\) is the minimum number of digits required to represent the exponent.

6 The sign in the exponent is produced. A plus sign is produced if the exponent value is zero.

NOTE 13.12

Examples:

<table>
<thead>
<tr>
<th>Internal Value</th>
<th>Output field Using SS, EN12.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.421</td>
<td>6.421E+00</td>
</tr>
<tr>
<td>-.5</td>
<td>-500.000E-03</td>
</tr>
<tr>
<td>.00217</td>
<td>2.170E-03</td>
</tr>
<tr>
<td>4721.3</td>
<td>4.721E+03</td>
</tr>
</tbody>
</table>

13.7.2.3.5 ES editing

1 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 (R715) 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.

2 The forms of the edit descriptor are ES\(w.d\) and ES\(w.d\) E\(e\) indicating that the external field occupies \(w\) positions, except when \(w\) is zero in which case the processor selects the field width. The fractional part of the field contains \(d\) digits. If \(e\) is positive the exponent part contains \(e\) digits, otherwise it contains the minimum number of digits required to represent the exponent value.

3 The form and interpretation of the input field is the same as for F\(w.d\) editing (13.7.2.3.2).

4 For an internal value that is an IEEE infinity or NaN, the form of the output field is the same as for F\(w.d\).

5 For an internal value that is neither an IEEE infinity nor a NaN, the form of the output field is

\[
[±] \cdot y \cdot x_1x_2\ldots x_d\exp
\]

where:

- ± signifies a plus sign or a minus sign;
- \(y\) is a decimal digit representative of the most significant digit of the internal value after rounding (13.7.2.3.8);
- . signifies a decimal symbol (13.6);
- \(x_1x_2\ldots x_d\) are the \(d\) next most significant digits of the internal value after rounding;
- \(\exp\) is a decimal exponent having one of the forms specified in table 13.3.

Table 13.3: ES exponent forms

<table>
<thead>
<tr>
<th>Edit Descriptor</th>
<th>Absolute Value of Exponent</th>
<th>Form of Exponent¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>ES(w.d)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>\exp</td>
<td>\leq 99)</td>
</tr>
<tr>
<td>99 &lt; (</td>
<td>\exp</td>
<td>\leq 999)</td>
</tr>
<tr>
<td>ES(w.d) E(e) with (e &gt; 0)</td>
<td>(</td>
<td>\exp</td>
</tr>
<tr>
<td>ES(w.d) E0</td>
<td>any</td>
<td>E±(z_1z_2\ldots z_s)</td>
</tr>
</tbody>
</table>

ⁱ(1) where each \(z\) is a digit, and \(s\) is the minimum number of digits required to represent the exponent.
NOTE 13.13

<table>
<thead>
<tr>
<th>Internal Value</th>
<th>Output field Using SS, ES12.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.421</td>
<td>6.421E+00</td>
</tr>
<tr>
<td>-.5</td>
<td>-5.000E-01</td>
</tr>
<tr>
<td>.00217</td>
<td>2.170E-03</td>
</tr>
<tr>
<td>4721.3</td>
<td>4.721E+03</td>
</tr>
</tbody>
</table>

13.7.2.3.6 EX editing

1 The EX edit descriptor produces an output field in the form of a hexadecimal-significand number.

2 The EX\(w.d\) and EX\(w.dEe\) edit descriptors indicate that the external field occupies \(w\) positions, except when \(w\) is zero in which case the processor selects the field width. The fractional part of the field contains \(d\) hexadecimal digits, except when \(d\) is zero in which case the processor selects the number of hexadecimal digits to be the minimum required so that the output field is equal to the internal value; \(d\) shall not be zero if the radix of the internal value is not a power of two. The hexadecimal point, represented by a decimal symbol, appears after the first hexadecimal digit. For the form EX\(w.d\), and for EX\(w.dE0\), the exponent part contains the minimum number of digits needed to represent the exponent; otherwise the exponent contains \(e\) digits. The \(e\) has no effect on input. The scale factor has no effect on output.

3 The form and interpretation of the input field is the same as for F\(w.d\) editing (13.7.2.3.2).

4 For an internal value that is an IEEE infinity or NaN, the form of the output field is the same as for F\(w.d\).

5 For an internal value that is neither an IEEE infinity nor a NaN, the form of the output field is

\[
[\pm] x_0 \cdot x_1 x_2 \ldots \exp
\]

where:

- \(\pm\) signifies a plus sign or a minus sign;
- \(\cdot\) signifies a decimal symbol (13.6);
- \(x_0 x_1 x_2 \ldots\) are the most significant hexadecimal digits of the internal value, after rounding if \(d\) is not zero (13.7.2.3.8);
- \(\exp\) is a binary exponent expressed as a decimal integer; for EX\(w.d\) and EX\(w.dE0\), the form is \(P \pm z_1 \ldots z_n\), where \(n\) is the minimum number of digits needed to represent \(\exp\), and for EX\(w.dEe\) with \(e\) greater than zero the form is \(P \pm z_1 \ldots z_e\).

6 The sign in the exponent is produced. A plus sign is produced if the exponent value is zero.

NOTE 13.14

<table>
<thead>
<tr>
<th>Internal value</th>
<th>Edit descriptor</th>
<th>Possible output with SS in effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.375</td>
<td>EX0.1</td>
<td>0X1.6P+0</td>
</tr>
<tr>
<td>-15.625</td>
<td>EX14.4E3</td>
<td>-0X1.F400P+003</td>
</tr>
<tr>
<td>1048580.0</td>
<td>EX0.0</td>
<td>0X1.00003P+20</td>
</tr>
</tbody>
</table>

13.7.2.3.7 Complex editing

1 A complex datum consists of a pair of separate real data. The editing of a scalar datum of complex type is specified by two edit descriptors each of which specifies the editing of real data. The first edit descriptor specifies the editing for the real part; the second specifies it for the imaginary part. The two edit descriptors may be different. Control and character string edit descriptors may be processed between the edit descriptor for the real part and the edit descriptor for the imaginary part.
### 13.7.2.3.8 Input/output rounding mode

1. The input/output rounding mode can be specified by an OPEN statement (12.5.2), a data transfer statement (12.6.2.13), or an edit descriptor (13.8.7).

2. In what follows, the term “decimal value” means the exact decimal number as given by the character string, while the term “internal value” means the number actually stored in the processor. For example, in dealing with the decimal constant 0.1, the decimal value is the mathematical quantity 1/10, which has no exact representation in binary form. Formatted output of real data involves conversion from an internal value to a decimal value; formatted input involves conversion from a decimal value to an internal value.

3. When the input/output rounding mode is UP, the value resulting from conversion shall be the smallest representable value that is greater than or equal to the original value. When the input/output rounding mode is DOWN, the value resulting from conversion shall be the largest representable value that is less than or equal to the original value. When the input/output rounding mode is ZERO, the value resulting from conversion shall be the closer of the two nearest representable values if one is closer than the other. If the two nearest representable values are equidistant from the original value, it is processor dependent which one of them is chosen. When the input/output rounding mode is NEAREST, the value resulting from conversion shall be the closer of the two nearest representable values or the value away from zero if halfway between them. When the input/output rounding mode is PROCESSOR_DEFINED, rounding during conversion shall be a processor-dependent default mode, which may correspond to one of the other modes.

4. On processors that support IEEE rounding on conversions (17.4), NEAREST shall correspond to round to nearest, as specified in ISO/IEC/IEEE 60559:2011.

#### NOTE 13.15

On processors that support IEEE rounding on conversions, the input/output rounding modes COMPATIBLE and NEAREST will produce the same results except when the datum is halfway between the two representable values. In that case, NEAREST will pick the even value, but COMPATIBLE will pick the value away from zero. The input/output rounding modes UP, DOWN, and ZERO have the same effect as those specified in ISO/IEC/IEEE 60559:2011 for round toward $+\infty$, round toward $-\infty$, and round toward 0, respectively.

### 13.7.2.4 B, O, and Z editing

1. The $B_w$, $B_w.m$, $O_w$, $O_w.m$, $Z_w$, and $Z_w.m$ edit descriptors indicate that the field to be edited occupies $w$ positions, except when $w$ is zero. When $w$ is zero, the processor selects the field width. On input, $w$ shall not be zero. The corresponding input/output list item shall be of type integer, real, or complex.

2. On input, $m$ has no effect.

3. In the standard form of the input field for the B, O, and Z edit descriptors the character string consists of binary, octal, or hexadecimal digits (as in R765, R766, R767) in the respective input field. The lower-case hexadecimal digits a through f in a hexadecimal input field are equivalent to the corresponding upper-case hexadecimal digits. If the input field does not have the standard form, and is not acceptable to the processor, an error condition occurs.

4. The value is INT (X) if the input list item is of type integer and REAL (X) if the input list item is of type real or complex, where X is a boz-literal-constant that specifies the same bit sequence as the digits of the input field.

5. The output field for the $B_w$, $O_w$, and $Z_w$ descriptors consists of zero or more leading blanks followed by the internal value in a form identical to the digits of a binary, octal, or hexadecimal constant, respectively, that specifies the same bit sequence but without leading zero bits.
NOTE 13.16
A binary, octal, or hexadecimal constant always consists of at least one digit or hexadecimal digit.

R1322 \texttt{hex-digit-string} \texttt{is \ hex-digit [ \ hex-digit ] ...}

The output field for the B\texttt{w.m}, O\texttt{w.m}, and Z\texttt{w.m} edit descriptor is the same as for the B\texttt{w}, O\texttt{w}, and Z\texttt{w} edit descriptor, except that the \texttt{digit-string} or \texttt{hex-digit-string} consists of at least \texttt{m} digits. If necessary, sufficient leading zeros are included to achieve the minimum of \texttt{m} digits. The value of \texttt{m} shall not exceed the value of \texttt{w}, except when \texttt{w} is zero. If \texttt{m} is zero and the internal value consists of all zero bits, the output field consists of only blank characters. When \texttt{m} and \texttt{w} are both zero, and the internal value consists of all zero bits, one blank character is produced.

13.7.3 Logical editing

The \texttt{Lw} edit descriptor indicates that the field occupies \texttt{w} positions. The specified input/output list item shall be of type logical. The G edit descriptor also may be used to edit logical data (13.7.5.3).

The standard form of the input field consists of optional blanks, optionally followed by a period, followed by a \texttt{T} for true or \texttt{F} for false. The \texttt{T} or \texttt{F} may be followed by additional characters in the field, which are ignored. If the input field does not have the standard form, and is not acceptable to the processor, an error condition occurs.

A lower-case letter is equivalent to the corresponding upper-case letter in a logical input field.

NOTE 13.17
The logical constants \texttt{.TRUE.} and \texttt{.FALSE.} are acceptable input forms.

The output field consists of \texttt{w−1} blanks followed by a \texttt{T} or \texttt{F}, depending on whether the internal value is true or false, respectively.

13.7.4 Character editing

The \texttt{A[w]} edit descriptor is used with an input/output list item of type character. The G edit descriptor also may be used to edit character data (13.7.5.4). The kind type parameter of all characters transferred and converted under control of one A or G edit descriptor is implied by the kind of the corresponding list item.

If a field width \texttt{w} is specified with the A edit descriptor, the field consists of \texttt{w} characters. If a field width \texttt{w} is not specified with the A edit descriptor, the number of characters in the field is the length of the corresponding list item, regardless of the value of the kind type parameter.

Let \texttt{len} be the length of the input/output list item. If the specified field width \texttt{w} for an A edit descriptor corresponding to an input item is greater than or equal to \texttt{len}, the rightmost \texttt{len} characters will be taken from the input field. If the specified field width \texttt{w} is less than \texttt{len}, the \texttt{w} characters will appear left justified with \texttt{len−w} trailing blanks in the internal value.

If the specified field width \texttt{w} for an A edit descriptor corresponding to an output item is greater than \texttt{len}, the output field will consist of \texttt{w−len} blanks followed by the \texttt{len} characters from the internal value. If the specified field width \texttt{w} is less than or equal to \texttt{len}, the output field will consist of the leftmost \texttt{w} characters from the internal value.

NOTE 13.18
For nondefault character kinds, the blank padding character is processor dependent.

If the file is connected for stream access, the output may be split across more than one record if it contains newline characters. A newline character is a nonblank character returned by the intrinsic function \texttt{NEW_LINE}. Beginning with the first character of the output field, each character that is not a newline is written to the current record in successive positions; each newline character causes file positioning at that point as if by slash editing.
(the current record is terminated at that point, a new empty record is created following the current record, this
new record becomes the last and current record of the file, and the file is positioned at the beginning of this new
record).

NOTE 13.19
If the intrinsic function NEW_LINE returns a blank character for a particular character kind, then the
processor does not support using a character of that kind to cause record termination in a formatted stream
file.

13.7.5 Generalized editing

13.7.5.1 Overview

1 The Gw, Gw.d and Gw.d Ee edit descriptors are used with an input/output list item of any intrinsic type. When
w is nonzero, these edit descriptors indicate that the external field occupies w positions. For real or complex
data the fractional part consists of a maximum of d digits and the exponent part consists of e digits. When these
edit descriptors are used to specify the input/output of integer, logical, or character data, d and e have no effect.
When w is zero the processor selects the field width. On input, w shall not be zero.

13.7.5.2 Generalized numeric editing

1 When used to specify the input/output of integer, real, and complex data, the Gw, Gw.d and Gw.d Ee edit
descriptors follow the general rules for numeric editing (13.7.2).

NOTE 13.20
The Gw.d Ee edit descriptor follows any additional rules for the Ew.d Ee edit descriptor.

13.7.5.2.1 Generalized integer editing

1 When used to specify the input/output of integer data, the Gw.d and Gw.d Ee edit descriptors follow the rules
for the Iw edit descriptor (13.7.2.2), except that w shall not be zero. When used to specify the output of integer
data, the G0 and G0.d edit descriptors follow the rules for the I0 edit descriptor.

13.7.5.2.2 Generalized real and complex editing

1 The form and interpretation of the input field is the same as for Fw.d editing (13.7.2.3.2).

2 If d is zero, kPEw.0 or kPEw.0Ee editing is used for Gw.0 editing or Gw.0Ee editing respectively.

3 When used to specify the output of real or complex data that is not an IEEE infinity or NaN, the G0 and G0.d
edit descriptors follow the rules for the Gw.dEe edit descriptor, except that any leading or trailing blanks are
removed. Reasonable processor-dependent values of w, d (if not specified), and e are used with each output value.

4 For an internal value that is an IEEE infinity or NaN, the form of the output field for the Gw.d and Gw.d Ee
edit descriptors is the same as for Fw.d, and the form of the output field for the G0 and G0.d edit descriptors is
the same as for F0.0.

5 Otherwise, the method of representation in the output field depends on the magnitude of the internal value
being edited. If the internal value is zero, let s be one. If the internal value is a number other than zero, let N
be the decimal value that is the result of converting the internal value to d significant digits according to the
input/output rounding mode and let s be the integer such that $10^{s-1} \leq |N| < 10^s$. If s < 0 or s > d, kPEw.d or
kPEw.dEe editing is used for Gw.d editing or Gw.dEe editing respectively, where k is the scale factor (13.8.5).
If 0 ≤ s ≤ d, the scale factor has no effect and F(w−n).(d−s),n(‘b’) editing is used where b is a blank and n is
4 for Gw.d editing, e + 2 for Gw.dEe editing if e > 0, and 4 for Gw.dE0 editing.

6 The value of w−n shall be positive.
NOTE 13.21

The scale factor has no effect on output unless the magnitude of the datum to be edited is outside the range that permits effective use of F editing.

13.7.5.3 Generalized logical editing

1. When used to specify the input/output of logical data, the $G_{w.d}$ and $G_{w.d} Ee$ edit descriptors follow the rules for the $Lw$ edit descriptor (13.7.3). When used to specify the output of logical data, the $G0$ and $G0.d$ edit descriptors follow the rules for the $L1$ edit descriptor.

13.7.5.4 Generalized character editing

1. When used to specify the input/output of character data, the $G_{w.d}$ and $G_{w.d} Ee$ edit descriptors follow the rules for the $A_w$ edit descriptor (13.7.4). When used to specify the output of character data, the $G0$ and $G0.d$ edit descriptors follow the rules for the $A$ edit descriptor with no field width.

13.7.6 User-defined derived-type editing

1. The DT edit descriptor specifies that a user-provided procedure shall be used instead of the processor’s default input/output formatting for processing a list item of derived type.

2. The DT edit descriptor may include a character literal constant. The character value “DT” concatenated with the character literal constant is passed to the defined input/output procedure as the iotype argument (12.6.4.8). The $v$ values of the edit descriptor are passed to the defined input/output procedure as the $v_list$ array argument.

NOTE 13.22

For the edit descriptor DT’Link List’(10, 4, 2), iotype is "DTLink List" and v_list is [10, 4, 2].

3. If a derived-type variable or value corresponds to a DT edit descriptor, there shall be an accessible interface to a corresponding defined input/output procedure for that derived type (12.6.4.8). A DT edit descriptor shall not correspond to a list item that is not of a derived type.

13.8 Control edit descriptors

13.8.1 Position edit descriptors

13.8.1.1 Position editing

1. The position edit descriptors T, TL, TR, and X, specify the position at which the next character will be transmitted to or from the record. If any character skipped by a position edit descriptor is of type nondefault character, and the unit is a default character internal file or an external non-Unicode file, the result of that position editing is processor dependent.

2. On input, if the position specified by a position edit descriptor is before the current position, portions of a record can be processed more than once, possibly with different editing.

3. On input, a position beyond the last character of the record may be specified if no characters are transmitted from such positions.

4. On output, a position edit descriptor does not by itself cause characters to be transmitted and therefore does not by itself affect the length of the record. If characters are transmitted to positions at or after the position specified by a position edit descriptor, positions skipped and not previously filled are filled with blanks. The result is as if the entire record were initially filled with blanks.

5. On output, a character in the record can be replaced. A position edit descriptor never directly causes a character
already placed in the record to be replaced, but it might result in positioning such that subsequent editing causes a replacement.

13.8.1.2 T, TL, and TR editing

1 The left tab limit affects file positioning by the T and TL edit descriptors. Immediately prior to nonchild data transfer (12.6.4.8.3), the left tab limit becomes defined as the character position of the current record or the current position of the stream file. If, during data transfer, the file is positioned to another record, the left tab limit becomes defined as character position one of that record.

2 The $T_n$ edit descriptor indicates that the transmission of the next character to or from a record is to occur at the $n$th character position of the record, relative to the left tab limit. This position can be in either direction from the current position.

3 The $TL_n$ edit descriptor indicates that the transmission of the next character to or from the record is to occur at the character position $n$ characters backward from the current position. However, if $n$ is greater than the difference between the current position and the left tab limit, the $TL_n$ edit descriptor indicates that the transmission of the next character to or from the record is to occur at the left tab limit.

4 The $TR_n$ edit descriptor indicates that the transmission of the next character to or from the record is to occur at the character position $n$ characters forward from the current position.

13.8.1.3 X editing

1 The $nX$ edit descriptor indicates that the transmission of the next character to or from a record is to occur at the character position $n$ characters forward from the current position.

NOTE 13.23

An $nX$ edit descriptor has the same effect as a $TR_n$ edit descriptor.

13.8.2 Slash editing

1 The slash edit descriptor indicates the end of data transfer to or from the current record.

2 On input from a file connected for sequential or stream access, the remaining portion of the current record is skipped and the file is positioned at the beginning of the next record. This record becomes the current record. On output to a file connected for sequential or stream access, a new empty record is created following the current record; this new record then becomes the last and current record of the file and the file is positioned at the beginning of this new record.

3 For a file connected for direct access, the record number is increased by one and the file is positioned at the beginning of the record that has that record number, if there is such a record, and this record becomes the current record.

NOTE 13.24

A record that contains no characters can be written on output. If the file is an internal file or a file connected for direct access, the record is filled with blank characters.

An entire record can be skipped on input.

4 The repeat specification is optional in the slash edit descriptor. If it is not specified, the default value is one.

13.8.3 Colon editing

1 The colon edit descriptor terminates format control if there are no more effective items in the input/output list (12.6.3). The colon edit descriptor has no effect if there are more effective items in the input/output list.
13.8.4 SS, SP, and S editing

1 The SS, SP, and S edit descriptors temporarily change (12.5.2) the sign mode (12.5.6.17, 12.6.2.14) for the connection. The edit descriptors SS, SP, and S set the sign mode corresponding to the SIGN= specifier values SUPPRESS, PLUS, and PROCESSOR_DEFINED, respectively.

2 The sign mode controls optional plus characters in numeric output fields. When the sign mode is PLUS, the processor shall produce a plus sign in any position that normally contains an optional plus sign. When the sign mode is SUPPRESS, the processor shall not produce a plus sign in such positions. When the sign mode is PROCESSOR_DEFINED, the processor has the option of producing a plus sign or not in such positions, subject to 13.7.2(5).

3 The SS, SP, and S edit descriptors affect only I, F, E, EN, ES, D, and G editing during the execution of an output statement. The SS, SP, and S edit descriptors have no effect during the execution of an input statement.

13.8.5 P editing

1 The $k$P edit descriptor temporarily changes (12.5.2) the scale factor for the connection to $k$. The scale factor affects the editing done by the F, E, EN, ES, EX, D, and G edit descriptors for numeric quantities.

2 The scale factor $k$ affects the appropriate editing in the following manner.

- On input, with F, E, EN, ES, EX, D, and G editing (provided that no exponent exists in the field), the effect is that the externally represented number equals the internally represented number multiplied by $10^k$; the scale factor is applied to the external decimal value and then this is converted using the input/output rounding mode.
- On input, with F, E, EN, ES, EX, D, and G editing, the scale factor has no effect if there is an exponent in the field.
- On output, with F output editing, the effect is that the externally represented number equals the internally represented number multiplied by $10^k$; the internal value is converted using the input/output rounding mode and then the scale factor is applied to the converted decimal value.
- On output, with E and D editing, the effect is that the significand (R715) part of the quantity to be produced is multiplied by $10^k$ and the exponent is reduced by $k$.
- On output, with G editing, the effect is suspended unless the magnitude of the datum to be edited is outside the range that permits the use of F editing. If the use of E editing is required, the scale factor has the same effect as with E output editing.
- On output, with EN, ES, and EX editing, the scale factor has no effect.

13.8.6 BN and BZ editing

1 The BN and BZ edit descriptors temporarily change (12.5.2) the blank interpretation mode (12.5.6.6, 12.6.2.6) for the connection. The edit descriptors BN and BZ set the blank interpretation mode corresponding to the BLANK= specifier values NULL and ZERO, respectively.

2 The blank interpretation mode controls the interpretation of nonleading blanks in numeric input fields. Such blank characters are interpreted as zeros when the blank interpretation mode has the value ZERO; they are ignored when the blank interpretation mode has the value NULL. The effect of ignoring blanks is to treat the input field as if blanks had been removed, the remaining portion of the field right justified, and the blanks replaced as leading blanks. However, a field containing only blanks has the value zero.

3 The blank interpretation mode affects only numeric editing (13.7.2) and generalized numeric editing (13.7.5.2) on input. It has no effect on output.

13.8.7 RU, RD, RZ, RN, RC, and RP editing

1 The round edit descriptors temporarily change (12.5.2) the connection’s input/output rounding mode (12.5.6.16, 12.6.2.13, 13.7.2.3.8). The round edit descriptors RU, RD, RZ, RN, RC, and RP set the input/output rounding
mode corresponding to the `ROUND=` specifier values UP, DOWN, ZERO, NEAREST, COMPATIBLE, and PROCESSOR_DEFINED, respectively. The input/output rounding mode affects the conversion of real and complex values in formatted input/output. It affects only D, E, EN, ES, F, and G editing.

13.8.8 DC and DP editing

1 The decimal edit descriptors temporarily change (12.5.2) the decimal edit mode (12.5.6.7, 12.6.2.7, 13.6) for the connection. The edit descriptors DC and DP set the decimal edit mode corresponding to the DECIMAL= specifier values COMMA and POINT, respectively.

2 The decimal edit mode controls the representation of the decimal symbol (13.6) during conversion of real and complex values in formatted input/output. The decimal edit mode affects only D, E, EN, ES, F, and G editing.

13.9 Character string edit descriptors

1 A character string edit descriptor shall not be used on input.

2 The character string edit descriptor causes characters to be written from the enclosed characters of the edit descriptor itself, including blanks. For a character string edit descriptor, the width of the field is the number of characters between the delimiting characters. Within the field, two consecutive delimiting characters are counted as a single character.

NOTE 13.25
A delimiter for a character string edit descriptor is either an apostrophe or quote.

13.10 List-directed formatting

13.10.1 Purpose of list-directed formatting

1 List-directed input/output allows data editing according to the type of the list item instead of by a format specification. It also allows data to be free-field, that is, separated by commas (or semicolons) or blanks.

13.10.2 Values and value separators

1 The characters in one or more list-directed records constitute a sequence of values and value separators. The end of a record has the same effect as a blank character, unless it is within a character constant. Any sequence of two or more consecutive blanks is treated as a single blank, unless it is within a character constant.

2 Each value is either a null value, \( c \), \( r^*c \), or \( r^* \), where \( c \) is a literal constant, optionally signed if integer or real, or an undelimited character constant and \( r \) is an unsigned, nonzero, integer literal constant. Neither \( c \) nor \( r \) shall have kind type parameters specified. The constant \( c \) is interpreted as though it had the same kind type parameter as the corresponding list item. The \( r^*c \) form is equivalent to \( r \) successive appearances of the constant \( c \), and the \( r^* \) form is equivalent to \( r \) successive appearances of the null value. Neither of these forms may contain embedded blanks, except where permitted within the constant \( c \).

3 A value separator is

- a comma optionally preceded by one or more contiguous blanks and optionally followed by one or more contiguous blanks, unless the decimal edit mode is COMMA, in which case a semicolon is used in place of the comma,
- a slash optionally preceded by one or more contiguous blanks and optionally followed by one or more contiguous blanks, or
- one or more contiguous blanks between two nonblank values or following the last nonblank value, where a nonblank value is a constant, an \( r^*c \) form, or an \( r^* \) form.
NOTE 13.26
Although a slash encountered in an input record is referred to as a separator, it actually causes termination
of list-directed and namelist input statements; it does not actually separate two values.

NOTE 13.27
If no list items are specified in a list-directed input/output statement, one input record is skipped or one
empty output record is written.

13.10.3 List-directed input
13.10.3.1 List-directed input forms

1 Input forms acceptable to edit descriptors for a given type are acceptable for list-directed formatting, except as
noted below. If the form of the input value is not acceptable to the processor for the type of the next effective
item in the list, an error condition occurs. Blanks are never used as zeros, and embedded blanks are not permitted
in constants, except within character constants and complex constants as specified below.

2 For the \( r^*c \) form of an input value, the constant \( c \) is interpreted as an undelimited character constant if the first
list item corresponding to this value is default, ASCII, or ISO 10646 character, there is a nonblank character
immediately after \( r^* \), and that character is not an apostrophe or a quotation mark; otherwise, \( c \) is interpreted as
a literal constant.

NOTE 13.28
The end of a record has the effect of a blank, except when it appears within a character constant.

3 When the next effective item is of type integer, the value in the input record is interpreted as if an \( I^w \) edit
descriptor with a suitable value of \( w \) were used.

4 When the next effective item is of type real, the input form is that of a numeric input field. A numeric input field
is a field suitable for F editing (13.7.2.3.2) that is assumed to have no fractional digits unless a decimal symbol
appears within the field.

5 When the next effective item is of type complex, the input form consists of a left parenthesis followed by an
ordered pair of numeric input fields separated by a comma (if the decimal edit mode is POINT) or semicolon
(if the decimal edit mode is COMMA), and followed by a right parenthesis. The first numeric input field is the
real part of the complex constant and the second is the imaginary part. Each of the numeric input fields may be
preceded or followed by any number of blanks and ends of records. The end of a record may occur after the real
part or before the imaginary part.

6 When the next effective item is of type logical, the input form shall not include value separators among the
optional characters permitted for L editing.

7 When the next effective item is of type character, the input form consists of a possibly delimited sequence of zero
or more rep-chars whose kind type parameter is implied by the kind of the effective item. Character sequences
may be continued from the end of one record to the beginning of the next record, but the end of record shall
not occur between a doubled apostrophe in an apostrophe-delimited character sequence, nor between a doubled
quote in a quote-delimited character sequence. The end of the record does not cause a blank or any other
character to become part of the character sequence. The character sequence may be continued on as many
records as needed. The characters blank, comma, semicolon, and slash may appear in default, ASCII, or ISO
10646 character sequences.

8 If the next effective item is default, ASCII, or ISO 10646 character and

- the character sequence does not contain value separators,
- the character sequence does not cross a record boundary,
- the first nonblank character is not a quotation mark or an apostrophe,
• the leading characters are not digits followed by an asterisk, and
• the character sequence contains at least one character,
the delimiting apostrophes or quotation marks are not required. If the delimiters are omitted, the character
sequence is terminated by the first blank, comma (if the decimal edit mode is POINT), semicolon (if the decimal
edit mode is COMMA), slash, or end of record; in this case apostrophes and quotation marks within the datum
are not to be doubled.

Let \( len \) be the current length of the next effective item, and let \( w \) be the length of the character sequence. If \( len \)
is less than or equal to \( w \), the leftmost \( len \) characters of the sequence are transmitted to the next effective item.
If \( len \) is greater than \( w \), the sequence is transmitted to the leftmost \( w \) characters of the next effective item and
the remaining \( len - w \) characters of the next effective item are filled with blanks.

NOTE 13.29
An allocatable, deferred-length character effective item does not have its allocation status or allocated
length changed as a result of list-directed input.

13.10.3.2 Null values

1 A null value is specified by

• the \( r^* \) form,
• no characters between consecutive value separators, or
• no characters before the first value separator in the first record read by each execution of a list-directed
input statement.

NOTE 13.30
The end of a record following any other value separator, with or without separating blanks, does not specify
a null value in list-directed input.

2 A null value has no effect on the definition status of the next effective item. A null value shall not be used for
either the real or imaginary part of a complex constant, but a single null value may represent an entire complex
constant.

3 A slash encountered as a value separator during execution of a list-directed input statement causes termination
of execution of that input statement after the transference of the previous value. Any characters remaining in the
current record are ignored. If there are additional items in the input list, the effect is as if null values had been
supplied for them. Any do-variable in the input list becomes defined as if enough null values had been supplied
for any remaining input list items.

NOTE 13.31
All blanks encountered during list-directed input are considered to be part of some value separator except for

• blanks embedded in a character sequence,
• embedded blanks surrounding the real or imaginary part of a complex constant, and
• leading blanks in the first record read by each execution of a list-directed input statement, unless
immediately followed by a slash or comma.

NOTE 13.32
List-directed input example:

INTEGER I; REAL X (8); CHARACTER (11) P;
NOTE 13.32 (cont.)

```fortran
COMPLEX Z; LOGICAL G
...
READ *, I, X, P, Z, G
...
```

The input data records are:

```
12345,12345,,2*1.5,4*
ISN'T_BOB'S,(123,0),.TEXAS$
```

The results are:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>12345</td>
</tr>
<tr>
<td>X (1)</td>
<td>12345.0</td>
</tr>
<tr>
<td>X (2)</td>
<td>unchanged</td>
</tr>
<tr>
<td>X (3)</td>
<td>1.5</td>
</tr>
<tr>
<td>X (4)</td>
<td>1.5</td>
</tr>
<tr>
<td>X (5) − X (8)</td>
<td>unchanged</td>
</tr>
<tr>
<td>P</td>
<td>ISN'T_BOB'S</td>
</tr>
<tr>
<td>Z</td>
<td>(123.0,0.0)</td>
</tr>
<tr>
<td>G</td>
<td>true</td>
</tr>
</tbody>
</table>

### 13.10.4 List-directed output

1. The form of the values produced is the same as that required for input, except as noted otherwise. With the exception of adjacent undelimited character sequences, the values are separated by one or more blanks or by a comma, or a semicolon if the decimal edit mode is COMMA, optionally preceded by one or more blanks and optionally followed by one or more blanks. Two undelimited character sequences are considered adjacent when both were written using list-directed input/output, no intervening data transfer or file positioning operations occurred on that unit occurred, and both were written either by a single data transfer statement, or during the execution of a parent data transfer statement along with its child data transfer statements. The form of the values produced by defined output (12.6.4.8) is determined by the defined output procedure; this form need not be compatible with list-directed input.

2. The processor may begin new records as necessary, but the end of record shall not occur within a constant except as specified for complex constants and character sequences. The processor shall not insert blanks within character sequences or within constants, except as specified for complex constants.

3. Logical output values are T for the value true and F for the value false.

4. Integer output constants are produced with the effect of an I\textsubscript{w} edit descriptor.

5. Real constants are produced with the effect of either an F edit descriptor or an E edit descriptor, depending on the magnitude \(x\) of the value and a range \(10^{d_1} \leq x < 10^{d_2}\), where \(d_1\) and \(d_2\) are processor-dependent integers. If the magnitude \(x\) is within this range or is zero, the constant is produced using 0PF\textsubscript{w.d}; otherwise, 1PE\textsubscript{w.d Ee} is used.

6. For numeric output, reasonable processor-dependent values of \(w\), \(d\), and \(e\) are used for each of the numeric constants output.

7. Complex constants are enclosed in parentheses with a separator between the real and imaginary parts, each produced as defined above for real constants. The separator is a comma if the decimal edit mode is POINT; it is a semicolon if the decimal edit mode is COMMA. The end of a record may occur between the separator and the imaginary part only if the entire constant is as long as, or longer than, an entire record. The only embedded blanks permitted within a complex constant are between the separator and the end of a record and one blank at
the beginning of the next record.

Character sequences produced when the delimiter mode has a value of NONE

- are not delimited by apostrophes or quotation marks,
- are not separated from each other by value separators,
- have each internal apostrophe or quotation mark represented externally by one apostrophe or quotation mark, and
- have a blank character inserted by the processor at the beginning of any record that begins with the 
  continuation of a character sequence from the preceding record.

Character sequences produced when the delimiter mode has a value of QUOTE are delimited by quotes, are 
preceded and followed by a value separator, and have each internal quote represented on the external medium by 
two contiguous quotes.

Character sequences produced when the delimiter mode has a value of APOSTROPHE are delimited by apo-
strophes, are preceded and followed by a value separator, and have each internal apostrophe represented on the 
external medium by two contiguous apostrophes.

If two or more successive values in an output record have identical values, the processor has the option of producing 
a repeated constant of the form \text{	extasciitilde{r}^*c} instead of the sequence of identical values.

Slashes, as value separators, and null values are not produced as output by list-directed formatting.

Except for new records created by explicit formatting within a defined output procedure or by continuation of 
delimited character sequences, each output record begins with a blank character.

\textbf{NOTE 13.33}

The length of the output records is not specified and is processor dependent.

\section*{13.11 Namelist formatting}

\subsection*{13.11.1 Purpose of namelist formatting}

Namelist input/output allows data editing with name-value subsequences. This facilitates documentation of input 
and output files and more flexibility on input.

\subsection*{13.11.2 Name-value subsequences}

The characters in one or more namelist records constitute a sequence of name-value subsequences, each of which 
consists of an \textit{object designator} followed by an equals and followed by one or more values and value separators. 
The equals may optionally be preceded or followed by one or more contiguous blanks. The end of a record has the 
same effect as a blank character, unless it is within a character constant. Any sequence of two or more consecutive 
blanks is treated as a single blank, unless it is within a character constant.

Each object designator shall begin with a name from the \textit{namelist-group-object-list} \textit{(8.9)} and shall follow the 
syntax of \textit{designator} \textit{(R901)}. It shall not contain a vector subscript or an \textit{image-selector} and shall not designate a 
zero-sized array, a zero-sized array section, or a zero-length character string. Each subscript, stride, and substring 
range expression shall be an optionally signed integer literal constant with no \textit{kind type parameter} specified. If 
a section subscript list appears, the number of section subscripts shall be equal to the rank of the object. If 
the namelist group object is of derived type, the designator in the input record may be either the name of the 
variable or the designator of one of its components, indicated by qualifying the variable name with the appropriate 
component name. Successive qualifications may be applied as appropriate to the shape and type of the variable 
represented. Each designator may be preceded and followed by one or more optional blanks but shall not contain 
embedded blanks.
A value separator for namelist formatting is the same as for list-directed formatting (13.10.2), or one or more contiguous blanks between a nonblank value and the following object designator or namelist comment (13.11.3.6).

13.11.3 Namelist input

13.11.3.1 Overall syntax

Input for a namelist input statement consists of

1. optional blanks and namelist comments,
2. the character & followed immediately by the namelist-group-name as specified in the NAMELIST statement,
3. one or more blanks,
4. a sequence of zero or more name-value subsequences separated by value separators, and
5. a slash to terminate the namelist input.

NOTE 13.34

A slash encountered in a namelist input record causes the input statement to terminate. A slash cannot be used to separate two values in a namelist input statement.

The order of the name-value subsequences in the input records need not match the order of the namelist-group-object-list. The input records need not specify all objects in the namelist-group-object-list. They may specify a part of an object more than once.

A group name or object name is without regard to case.

13.11.3.2 Namelist input processing

The name-value subsequences are evaluated serially, in left-to-right order. A namelist group object designator may appear in more than one name-value subsequence. The definition status of an object that is not a subobject of a designator in any name-value subsequence remains unchanged.

When the designator in the input record represents an array variable or a variable of derived type, the effect is as if the variable represented were expanded into a sequence of scalar list items, in the same way that formatted input/output list items are expanded (12.6.3). The number of values following the equals shall not exceed the number of list items in the expanded sequence, but may be less; in the latter case, the effect is as if sufficient null values had been appended to match any remaining list items in the expanded sequence. Except as noted elsewhere in this subclause, if an input value is not acceptable to the processor for the type of the list item in the corresponding position in the expanded sequence, an error condition occurs.

NOTE 13.35

For example, if the designator in the input record designates an integer array of size 100, at most 100 values, each of which is either a digit string or a null value, can follow the equals; these values would then be assigned to the elements of the array in array element order.

A slash encountered as a value separator during the execution of a namelist input statement causes termination of execution of that input statement after transference of the previous value. If there are additional items in the namelist group object being transferred, the effect is as if null values had been supplied for them.

Successive namelist records are read by namelist input until a slash is encountered; the remainder of the record is ignored.

A namelist comment may appear after any value separator except a slash (which terminates namelist input). A namelist comment is also permitted to start in the first nonblank position of an input record except within a character literal constant.
13.11.3.3 Namelist input values

1 Each value is either a null value (13.11.3.4), c, r*c, or r*, where c is a literal constant, optionally signed if integer or real, and r is an unsigned, nonzero, integer literal constant. A kind type parameter shall not be specified for c or r. The constant c is interpreted as though it had the same kind type parameter as the corresponding effective item. The r*c form is equivalent to r successive appearances of the constant c, and the r* form is equivalent to r successive null values. Neither of these forms may contain embedded blanks, except where permitted within the constant c.

2 The datum c (13.11) is any input value acceptable to format specifications for a given type, except for a restriction on the form of input values corresponding to list items of types logical, integer, and character as specified in this subclause. The form of a real or complex value is dependent on the decimal edit mode in effect (13.6). The form of an input value shall be acceptable for the type of the namelist group object list item. The number and forms of the input values that may follow the equals in a name-value subsequence depend on the shape and type of the object represented by the name in the input record. When the name in the input record is that of a scalar variable of an intrinsic type, the equals shall not be followed by more than one value. Blanks are never used as zeros, and embedded blanks are not permitted in constants except within character constants and complex constants as specified in this subclause.

3 When the next effective item is of type real, the input form of the input value is that of a numeric input field. A numeric input field is a field suitable for F editing (13.7.2.3.2) that is assumed to have no fractional digits unless a decimal symbol appears within the field.

4 When the next effective item is of type complex, the input form of the input value consists of a left parenthesis followed by an ordered pair of numeric input fields separated by a comma (if the decimal edit mode is POINT) or a semicolon (if the decimal edit mode is COMMA), and followed by a right parenthesis. The first numeric input field is the real part of the complex constant and the second field is the imaginary part. Each of the numeric input fields may be preceded or followed by any number of blanks and ends of records. The end of a record may occur between the real part and the comma or semicolon, or between the comma or semicolon and the imaginary part.

5 When the next effective item is of type logical, the input form of the input value shall not include equals or value separators among the optional characters permitted for L editing (13.7.3).

6 When the next effective item is of type integer, the value in the input record is interpreted as if an Iw edit descriptor with a suitable value of w were used.

7 When the next effective item is of type character, the input form consists of a sequence of zero or more rep-chars whose kind type parameter is implied by the kind of the corresponding list item, delimited by apostrophes or quotes. Such a sequence may be continued from the end of one record to the beginning of the next record, but the end of record shall not occur between a doubled apostrophe in an apostrophe-delimited sequence, nor between a doubled quote in a quote-delimited sequence. The end of the record does not cause a blank or any other character to become part of the sequence. The sequence may be continued on as many records as needed. The characters blank, comma, semicolon, and slash may appear in such character sequences.

NOTE 13.36
The delimiters in the input form for a namelist input item of type character avoid the ambiguity that could arise between undelimited character sequences and object names. The value of the DELIM= specifier, if any, in the OPEN statement for an external file is ignored during namelist input (12.5.6.8).

8 Let len be the length of the next effective item, and let w be the length of the character sequence. If len is less than or equal to w, the leftmost len characters of the sequence are transmitted to the next effective item. If len is greater than w, the constant is transmitted to the leftmost w characters of the next effective item and the remaining len−w characters of the next effective item are filled with blanks. The effect is as though the sequence were assigned to the next effective item in an intrinsic assignment statement (10.2.1.3).
13.11.3.4 Null values

1 A null value is specified by

   • the r* form,
   • blanks between two consecutive nonblank value separators following an equals,
   • a value separator that is the first nonblank character following an equals, or
   • two consecutive nonblank value separators.

2 A null value has no effect on the definition status of the corresponding input list item. If the namelist group object list item is defined, it retains its previous value; if it is undefined, it remains undefined. A null value shall not be used as either the real or imaginary part of a complex constant, but a single null value may represent an entire complex constant.

NOTE 13.37
The end of a record following a value separator, with or without intervening blanks, does not specify a null value in namelist input.

13.11.3.5 Blanks

1 All blanks in a namelist input record are considered to be part of some value separator except for

   • blanks embedded in a character constant,
   • embedded blanks surrounding the real or imaginary part of a complex constant,
   • leading blanks following the equals unless followed immediately by a slash or comma, or a semicolon if the decimal edit mode is COMMA, and
   • blanks between a name and the following equals.

13.11.3.6 Namelist comments

1 Except within a character literal constant, a “!” character after a value separator or in the first nonblank position of a namelist input record initiates a comment. The comment extends to the end of the record and may contain any graphic character in the processor-dependent character set. The comment is ignored. A slash within the namelist comment does not terminate execution of the namelist input statement. Namelist comments are not allowed in stream input because comments depend on record structure.

NOTE 13.38

Namelist input example:

```fortran
INTEGER I; REAL X (8); CHARACTER (11) P; COMPLEX Z; LOGICAL G
NAMELIST / TODAY / G, I, P, Z, X
READ (*, NML = TODAY)
```

The input data records are:

```text
&TODAY I = 12345, X(1) = 12345, X(3:4) = 2*1.5, I=6, ! This is a comment.
P = 'ISN'T_BOB'S', Z = (123,0)/
```

The results stored are:
NOTE 13.38 (cont.)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>6</td>
</tr>
<tr>
<td>X (1)</td>
<td>12345.0</td>
</tr>
<tr>
<td>X (2)</td>
<td>unchanged</td>
</tr>
<tr>
<td>X (3)</td>
<td>1.5</td>
</tr>
<tr>
<td>X (4)</td>
<td>1.5</td>
</tr>
<tr>
<td>X (5) − X (8)</td>
<td>unchanged</td>
</tr>
<tr>
<td>P</td>
<td>ISN’T_BOB’S</td>
</tr>
<tr>
<td>Z</td>
<td>(123.0,0.0)</td>
</tr>
<tr>
<td>G</td>
<td>unchanged</td>
</tr>
</tbody>
</table>

13.11.4 Namelist output

13.11.4.1 Form of namelist output

1 The form of the output produced by intrinsic namelist output shall be suitable for input, except for character output. The names in the output are in upper case. With the exception of adjacent undelimited character values, the values are separated by one or more blanks or by a comma, or a semicolon if the decimal edit mode is COMMA, optionally preceded by one or more blanks and optionally followed by one or more blanks. The form of the output produced by defined output (12.6.4.8) is determined by the defined output procedure; this form need not be compatible with namelist input.

2 Namelist output shall not include namelist comments.

3 The processor may begin new records as necessary. However, except for complex constants and character values, the end of a record shall not occur within a constant, character value, or name, and blanks shall not appear within a constant, character value, or name.

NOTE 13.39

The length of the output records is not specified exactly and is processor dependent.

13.11.4.2 Namelist output editing

1 Values in namelist output records are edited as for list-directed output (13.10.4).

NOTE 13.40

Namelist output records produced with a DELIM= specifier with a value of NONE and which contain a character sequence might not be acceptable as namelist input records.

13.11.4.3 Namelist output records

1 If two or more successive values for the same namelist group item in an output record produced have identical values, the processor has the option of producing a repeated constant of the form r* instead of the sequence of identical values.

2 The name of each namelist group object list item is placed in the output record followed by an equals and a list of values of the namelist group object list item.

3 An ampersand character followed immediately by a namelist-group-name is placed at the start of the first output record to indicate which particular group of data objects is being output. A slash is placed in the output record to indicate the end of the namelist formatting.

4 A null value is not produced by namelist formatting.

5 Except for new records created by explicit formatting within a defined output procedure or by continuation of
delimited character sequences, each output record begins with a blank character.
14 Program units

14.1 Main program

A Fortran main program is a program unit that does not contain a SUBROUTINE, FUNCTION, MODULE, SUBMODULE, or BLOCK DATA statement as its first statement.

R1401 main-program is

[ program-stmt ]
[ specification-part ]
[ execution-part ]
[ internal-subprogram-part ]
end-program-stmt

R1402 program-stmt is PROGRAM program-name

R1403 end-program-stmt is END [ PROGRAM [ program-name ] ]

C1401 (R1401) The program-name may be included in the end-program-stmt only if the optional program-stmt is used and, if included, shall be identical to the program-name specified in the program-stmt.

NOTE 14.1
The program name is global to the program (19.2). For explanatory information about uses for the program name, see subclause C.9.1.

NOTE 14.2
An example of a main program is:

PROGRAM ANALYZE

REAL A, B, C (10,10) ! Specification part
CALL FIND ! Execution part
CONTAINS

SUBROUTINE FIND ! Internal subprogram

... END SUBROUTINE FIND
END PROGRAM ANALYZE

The main program may be defined by means other than Fortran; in that case, the program shall not contain a main-program program unit.

A reference to a Fortran main-program shall not appear in any program unit in the program, including itself.

14.2 Modules

14.2.1 Module syntax and semantics

A module contains declarations, specifications, and definitions. Public identifiers of module entities are accessible to other program units by use association as specified in 14.2.2. A module that is provided as an inherent part of the processor is an intrinsic module. A nonintrinsic module is defined by a module program unit or a means other than Fortran.

Procedures and types defined in an intrinsic module are not themselves intrinsic.
module-stmt is

[ specification-part ]
[ module-subprogram-part ]
end-module-stmt

module-stmt is
MODULE module-name

end-module-stmt is
END [ MODULE [ module-name ] ]

contains-stmt is

[ module-subprogram ] ...

function-subprogram is

or subroutine-subprogram

or separate-module-subprogram

If the module-name is specified in the end-module-stmt, it shall be identical to the module-name specified in the module-stmt.

A module specification-part shall not contain a stmt-function-stmt, an entry-stmt, or a format-stmt.

If a procedure declared in the scoping unit of a module has an implicit interface, it shall be given the EXTERNAL attribute in that scoping unit; if it is a function, its type and type parameters shall be explicitly declared in a type declaration statement in that scoping unit.

If an intrinsic procedure is declared in the scoping unit of a module, it shall explicitly be given the INTRINSIC attribute in that scoping unit or be used as an intrinsic procedure in that scoping unit.

The module name is global to the program (19.2).

Although statement function definitions, ENTRY statements, and FORMAT statements cannot appear in the specification part of a module, they can appear in the specification part of a module subprogram in the module.

For a discussion of the impact of modules on dependent compilation, see subclause C.9.2.

For examples of the use of modules, see subclause C.9.3.

The USE statement specifies use association. A USE statement is a reference to the module it specifies. At the time a USE statement is processed, the public portions of the specified module shall be available. A module shall not reference itself, either directly or indirectly.

The USE statement provides the means by which a scoping unit accesses named data objects, derived types, procedures, abstract interfaces, generic identifiers, and namelist groups in a module. The entities in the scoping unit are use associated with the entities in the module. The accessed entities have the attributes specified in the module, except that a local entity may have a different accessibility attribute, it may have the ASYNCHRONOUS attribute even if the associated module entity does not, and if it is not a coarray it may have the VOLATILE attribute even if the associated module entity does not. The entities made accessible are identified by the names or generic identifiers used to identify them in the module. By default, the local entities are identified by the same identifiers in the scoping unit containing the USE statement, but it is possible to specify that different local
identifiers are used. A use-associated entity is considered to have been previously defined.

**NOTE 14.7**
The accessibility of module entities can be controlled by accessibility attributes (7.5.2.2, 8.5.2), and the ONLY option of the USE statement. Definability of module entities can be controlled by the PROTECTED attribute (8.5.15).

R1409 `use-stmt` is

```
USE [ , module-nature ] :: ] module-name [ , rename-list ]
or USE [ , module-nature ] :: ] module-name , ■
```

R1410 `module-nature` is

```
INTRINSIC
or NON_INTRINSIC
```

R1411 `rename` is

```
local-name => use-name
or OPERATOR (local-defined-operator) => ■
```

R1412 `only` is

```
generic-spec
or only-use-name
or rename
```

R1413 `only-use-name` is

```
use-name
```

C1404 (R1409) If `module-nature` is INTRINSIC, `module-name` shall be the name of an intrinsic module.

C1405 (R1409) If `module-nature` is NON_INTRINSIC, `module-name` shall be the name of a nonintrinsic module.

C1406 (R1409) A scoping unit shall not access an intrinsic module and a nonintrinsic module of the same name.

C1407 (R1411) OPERATOR (use-defined-operator) shall not identify a type-bound generic interface.

C1408 (R1412) The `generic-spec` shall not identify a type-bound generic interface.

**NOTE 14.8**
Constraints C1407 and C1408 do not prevent accessing a `generic-spec` that is declared by an interface block, even if a type-bound generic interface has the same `generic-spec`.

C1409 (R1412) Each `generic-spec` shall be a public entity in the module.

C1410 (R1413) Each `use-name` shall be the name of a public entity in the module.

R1414 `local-defined-operator` is

```
defined-unary-op
or defined-binary-op
```

R1415 `use-defined-operator` is

```
defined-unary-op
or defined-binary-op
```

C1411 (R1415) Each `use-defined-operator` shall be a public entity in the module.

3 A `use-stmt` without a `module-nature` provides access either to an intrinsic or to a nonintrinsic module. If the `module-name` is the name of both an intrinsic and a nonintrinsic module, the nonintrinsic module is accessed.

4 The USE statement without the ONLY option provides access to all public entities in the specified module.

5 A USE statement with the ONLY option provides access only to those entities that appear as `generic-specs`, `use-names`, or `use-defined-operators` in the `only-list`.

6 More than one USE statement for a given module may appear in a specification part. If one of the USE statements
is without an ONLY option, all public entities in the module are accessible. If all the USE statements have ONLY options, only those entities in one or more of the only-lists are accessible.

7 An accessible entity in the referenced module has one or more local identifiers. These identifiers are

- the identifier of the entity in the referenced module if that identifier appears as an only-use-name or as the defined-operator of a generic-spec in any only for that module,
- each of the local-names or local-defined-operators that the entity is given in any rename for that module, and
- the identifier of the entity in the referenced module if that identifier does not appear as a use-name or use-defined-operator in any rename for that module.

8 An ultimate entity is a module entity that is not accessed by use association. Two or more accessible ultimate entities, other than generic interfaces or defined operators, may have the same local identifier only if the identifier is not used. Generic interfaces and defined operators are handled as described in 15.4.3.4. Except for these cases, the local identifier of any entity given accessibility by a USE statement shall differ from the local identifiers of all other entities accessible to the scoping unit.

NOTE 14.9
There is no prohibition against a use-name or use-defined-operator appearing multiple times in one USE statement or in multiple USE statements involving the same module. As a result, it is possible for one use-associated entity to be accessible by more than one local identifier.

9 The local identifier of an entity made accessible by a USE statement shall not appear in any other nonexecutable statement that would cause any attribute (8.5) of the entity to be specified in the scoping unit that contains the USE statement, except that it may appear in a PUBLIC or PRIVATE statement in the scoping unit of a module and it may be given the ASYNCHRONOUS or VOLATILE attribute.

10 An entity in a scoping unit that is accessed by use association through more than one use path, has the ASYNCHRONOUS or VOLATILE attribute in any of those use paths, and is not given that attribute in that scoping unit, shall have that attribute in all use paths.

11 The appearance of such a local identifier in a PUBLIC statement in a module causes the entity accessible by the USE statement to be a public entity of that module. If the identifier appears in a PRIVATE statement in a module, the entity is not a public entity of that module. If the local identifier does not appear in either a PUBLIC or PRIVATE statement, it assumes the default accessibility attribute (8.6.1) of that scoping unit.

NOTE 14.10
The constraints in subclauses 8.10.1, 8.10.2, and 8.9 prohibit the local-name from appearing as a common-block-object in a COMMON statement, an equivalence-object in an EQUIVALENCE statement, or a namelist-group-name in a NAMELIST statement, respectively. There is no prohibition against the local-name appearing as a common-block-name or a namelist-group-object.

NOTE 14.11
For a discussion of the impact of the ONLY option and renaming on dependent compilation, see subclause C.9.2.1.

NOTE 14.12
Examples:

USE STATS_LIB

provides access to all public entities in the module STATS_LIB.

USE MATH_LIB; USE STATS_LIB, SPROD => PROD
NOTE 14.12 (cont.)

makes all public entities in both MATH_LIB and STATS_LIB accessible. If MATH_LIB contains an
entity called PROD, it is accessible by its own name while the entity PROD of STATS_LIB is accessible
by the name SPROD.

USE STATS_LIB, ONLY: YPROD; USE STATS_LIB, ONLY : PROD

makes public entities YPROD and PROD in STATS_LIB accessible.

USE STATS_LIB, ONLY : YPROD; USE STATS_LIB

makes all public entities in STATS_LIB accessible.

14.2.3 Submodules

A submodule is a program unit that extends a module or another submodule. The program unit that it extends
is its host, and is specified by the parent-identifier in the submodule-stmt.

A module or submodule is an ancestor program unit of all of its descendants, which are its submodules and their
descendants. The submodule identifier is the ordered pair whose first element is the ancestor module name and
whose second element is the submodule name; the submodule name by itself is not a local or global identifier.

NOTE 14.13

A module and its submodules stand in a tree-like relationship one to another, with the module at the root.
Therefore, a submodule has exactly one ancestor module and can have one or more ancestor submodules.

A submodule may provide implementations for separate module procedures (15.6.2.5), each of which is declared
(15.4.3.2) within that submodule or one of its ancestors, and declarations and definitions of other entities that
are accessible by host association in its descendants.

R1416 submodule is submodule-stmt
    [ specification-part ]
    [ module-subprogram-part ]
end-submodule-stmt

R1417 submodule-stmt is SUBMODULE (parent-identifier) submodule-name

R1418 parent-identifier is ancestor-module-name [ : parent-submodule-name ]

R1419 end-submodule-stmt is END [ SUBMODULE [ submodule-name ] ]

C1412 (R1416) A submodule specification-part shall not contain a format-stmt, entry-stmt, or stmt-function-stmt.

C1413 (R1418) The ancestor-module-name shall be the name of a nonintrinsic module that declares a separate
module procedure; the parent-submodule-name shall be the name of a descendant of that module.

C1414 (R1416) If a submodule-name appears in the end-submodule-stmt, it shall be identical to the one in the
submodule-stmt.

14.3 Block data program units

A block data program unit is used to provide initial values for data objects in named common blocks.

R1420 block-data is block-data-stmt
    [ specification-part ]
end-block-data-stmt
R1421  block-data-stmt is BLOCK DATA [ block-data-name ]

R1422  end-block-data-stmt is END [ BLOCK DATA [ block-data-name ] ]

C1415  (R1420) The block-data-name shall be included in the end-block-data-stmt only if it was provided in the block-data-stmt and, if included, shall be identical to the block-data-name in the block-data-stmt.

C1416  (R1420) A block-data specification-part shall contain only derived-type definitions and ASYNCHRONOUS, BIND, COMMON, DATA, DIMENSION, EQUIVALENCE, IMPLICIT, INTRINSIC, PARAMETER, POINTER, SAVE, TARGET, USE, VOLATILE, and type declaration statements.

C1417  (R1420) A type declaration statement in a block-data specification-part shall not contain ALLOCATABLE, EXTERNAL, or BIND attribute specifiers.

2  If an object in a named common block is initially defined, all storage units in the common block storage sequence shall be specified even if they are not all initially defined. More than one named common block may have objects initially defined in a single block data program unit.

3  Only an object in a named common block may be initially defined in a block data program unit.

4  The same named common block shall not be specified in more than one block data program unit in a program.

5  There shall not be more than one unnamed block data program unit in a program.
15 Procedures

15.1 Concepts

1 The concept of a procedure was introduced in 5.2.3. This clause contains a complete description of procedures.

The actions specified by a procedure are performed when the procedure is invoked by execution of a reference to it.

2 The sequence of actions encapsulated by a procedure has access to entities in the procedure reference by way of argument association (15.5.2). A name that appears as a dummy-arg-name in the SUBROUTINE, FUNCTION, or ENTRY statement in the declaration of a procedure (R1536) is a dummy argument. Dummy arguments are also specified for intrinsic procedures and procedures in intrinsic modules in Clauses 16, 17, and 18.

15.2 Procedure classifications

15.2.1 Procedure classification by reference

1 The definition of a procedure specifies it to be a function or a subroutine. A reference to a function either appears explicitly as a primary within an expression, or is implied by a defined operation (10.1.6) within an expression. A reference to a subroutine is a CALL statement, a defined assignment statement (10.2.1.4), the appearance of an object processed by defined input/output (12.6.4.8) in an input/output list, or finalization (7.5.6).

2 A procedure is classified as elemental if it is a procedure that may be referenced elementally (15.8).

15.2.2 Procedure classification by means of definition

15.2.2.1 Intrinsic procedures

1 A procedure that is provided as an inherent part of the processor is an intrinsic procedure.

15.2.2.2 External, internal, and module procedures

1 An external procedure is a procedure that is defined by an external subprogram or by a means other than Fortran.

2 An internal procedure is a procedure that is defined by an internal subprogram. Internal subprograms may appear in the main program, in an external subprogram, or in a module subprogram. Internal subprograms shall not appear in other internal subprograms. Internal subprograms are the same as external subprograms except that the name of the internal procedure is not a global identifier, an internal subprogram shall not contain an ENTRY statement, and the internal subprogram has access to host entities by host association.

3 A module procedure is a procedure that is defined by a module subprogram.

4 A subprogram defines a procedure for the SUBROUTINE or FUNCTION statement. If the subprogram has one or more ENTRY statements, it also defines a procedure for each of them.

15.2.2.3 Dummy procedures

1 A dummy argument that is specified to be a procedure or appears as the procedure designator in a procedure reference is a dummy procedure. A dummy procedure with the POINTER attribute is a dummy procedure pointer.
15.2.2.4 Procedure pointers
A procedure pointer is a procedure that has the EXTERNAL and POINTER attributes; it may be pointer associated with an external procedure, an internal procedure, an intrinsic procedure, a module procedure, or a dummy procedure that is not a procedure pointer.

15.2.2.5 Statement functions
A function that is defined by a single statement is a statement function (15.6.4).

15.3 Characteristics

15.3.1 Characteristics of procedures
The characteristics of a procedure are the classification of the procedure as a function or subroutine, whether it is pure, whether it is elemental, whether it has the BIND attribute, the characteristics of its dummy arguments, and the characteristics of its function result if it is a function.

15.3.2 Characteristics of dummy arguments
15.3.2.1 General
Each dummy argument has the characteristic that it is a dummy data object, a dummy procedure, or an asterisk (alternate return indicator).

15.3.2.2 Characteristics of dummy data objects
The characteristics of a dummy data object are its type, its type parameters (if any), its shape (unless it is assumed-rank), its corank, its codimensions, its intent (8.5.10, 8.6.9), whether it is optional (8.5.12, 8.6.10), whether it is allocatable (8.5.3), whether it has the ASYNCHRONOUS (8.5.4), CONTIGUOUS (8.5.7), VALUE (8.5.18), or VOLATILE (8.5.19) attributes, whether it is polymorphic, and whether it is a pointer (8.5.14, 8.6.12) or a target (8.5.17, 8.6.15). If a type parameter of an object or a bound of an array is not a constant expression, the exact dependence on the entities in the expression is a characteristic. If a rank, shape, size, type, or type parameter is assumed or deferred, it is a characteristic.

15.3.2.3 Characteristics of dummy procedures
The characteristics of a dummy procedure are the explicitness of its interface (15.4.2), its characteristics as a procedure if the interface is explicit, whether it is a pointer, and whether it is optional (8.5.12, 8.6.10).

15.3.2.4 Characteristics of asterisk dummy arguments
An asterisk as a dummy argument has no characteristics.

15.3.3 Characteristics of function results
The characteristics of a function result are its type, type parameters (if any), rank, whether it is polymorphic, whether it is allocatable, whether it is a pointer, whether it has the CONTIGUOUS attribute, and whether it is a procedure pointer. If a function result is an array that is not allocatable or a pointer, its shape is a characteristic. If a type parameter of a function result or a bound of a function result array is not a constant expression, the exact dependence on the entities in the expression is a characteristic. If type parameters of a function result are deferred, which parameters are deferred is a characteristic. Whether the length of a character function result is assumed is a characteristic.
15.4 Procedure interface

15.4.1 Interface and abstract interface

The interface of a procedure determines the forms of reference through which it may be invoked. The procedure’s interface consists of its name, binding label, generic identifiers, characteristics, and the names of its dummy arguments. The characteristics and binding label of a procedure are fixed, but the remainder of the interface may differ in differing contexts, except that for a separate module procedure body (15.6.2.5), the dummy argument names and whether it is recursive shall be the same as in its corresponding module procedure interface body (15.4.3.2).

An abstract interface is a set of procedure characteristics with the dummy argument names.

15.4.2 Implicit and explicit interfaces

15.4.2.1 Interfaces and scopes

The interface of a procedure is either explicit or implicit. It is explicit if it is

- an internal procedure, module procedure, or intrinsic procedure,
- a subroutine, or a function with a separate result name, within the scoping unit that defines it, or
- a procedure declared by a procedure declaration statement that specifies an explicit interface, or by an interface body.

Otherwise, the interface of the identifier is implicit. The interface of a statement function is always implicit.

NOTE 15.1
For example, the subroutine LLS of C.9.3.4 has an explicit interface.

15.4.2.2 Explicit interface

Within the scope of a procedure identifier, the procedure shall have an explicit interface if it is not a statement function and

(1) a reference to the procedure appears
   (a) with an argument keyword (15.5.2), or
   (b) in a context that requires it to be pure,

(2) the procedure has a dummy argument that
   (a) has the ALLOCATABLE, ASYNCHRONOUS, OPTIONAL, POINTER, TARGET, VALUE, or VOLATILE attribute,
   (b) is an assumed-shape array,
   (c) is assumed-rank,
   (d) is a coarray,
   (e) is of a parameterized derived type, or
   (f) is polymorphic,

(3) the procedure has a result that
   (a) is an array,
   (b) is a pointer or is allocatable, or
   (c) has a nonassumed type parameter value that is not a constant expression,

(4) the procedure is elemental, or

(5) the procedure has the BIND attribute.
15.4.3 Specification of the procedure interface

15.4.3.1 General

1 The interface for an internal, external, module, or dummy procedure is specified by a FUNCTION, SUBROUTINE, or ENTRY statement and by specification statements for the dummy arguments and the result of a function. These statements may appear in the procedure definition, in an interface body, or both, except that the ENTRY statement shall not appear in an interface body.

NOTE 15.2

An interface body cannot be used to describe the interface of an internal procedure, a module procedure that is not a separate module procedure, or an intrinsic procedure because the interfaces of such procedures are already explicit. However, the name of a procedure can appear in a PROCEDURE statement in an interface block (15.4.3.2).

15.4.3.2 Interface block

R1501 interface-block is interface-stmt

[ interface-specification ] ...

end-interface-stmt

R1502 interface-specification is interface-body

or procedure-stmt

R1503 interface-stmt is INTERFACE [ generic-spec ]

or ABSTRACT INTERFACE

R1504 end-interface-stmt is END INTERFACE [ generic-spec ]

R1505 interface-body is function-stmt

[ specification-part ]

end-function-stmt

or subroutine-stmt

[ specification-part ]

end-subroutine-stmt

R1506 procedure-stmt is [ MODULE ] PROCEDURE [ :: ] specific-procedure-list

R1507 specific-procedure is procedure-name

R1508 generic-spec is generic-name

or OPERATOR ( defined-operator )

or ASSIGNMENT ( = )

or defined-io-generic-spec

R1509 defined-io-generic-spec is READ (FORMATTED)

or READ (UNFORMATTED)

or WRITE (FORMATTED)

or WRITE (UNFORMATTED)

C1501 (R1501) An interface-block in a subprogram shall not contain an interface-body for a procedure defined by that subprogram.

C1502 (R1501) If the end-interface-stmt includes a generic-spec, the interface-stmt shall specify the same generic-spec, except that if one generic-spec has a defined-operator that is .LT., .LE., .GT., .GE., .EQ., or .NE., the other generic-spec may have a defined-operator that is the corresponding operator <, <=,

>., >=., ==, or /=.
C1503  (R1503) If the interface-stmt is ABSTRACT INTERFACE, then the function-name or subroutine-name in the function-stmt or subroutine-stmt shall not be the same as a keyword that specifies an intrinsic type.

C1504  (R1502) A procedure-stmt is allowed only in an interface block that has a generic-spec.

C1505  (R1505) An interface-body of a pure procedure shall specify the intents of all dummy arguments except alternate return indicators, dummy procedures, and arguments with the POINTER or VALUE attribute.

C1506  (R1505) An interface-body shall not contain a data-stmt, format-stmt, entry-stmt, or stmt-function-stmt.

C1507  (R1506) If MODULE appears in a procedure-stmt, each procedure-name in that statement shall denote a module procedure.

C1508  (R1507) A procedure-name shall denote a nonintrinsic procedure that has an explicit interface.

C1509  (R1501) An interface-specification in a generic interface block shall not specify a procedure that was specified previously in any accessible interface with the same generic identifier.

1 An external or module subprogram specifies a specific interface for each procedure defined in that subprogram.

2 An interface block introduced by ABSTRACT INTERFACE is an abstract interface block. An interface body in an abstract interface block specifies an abstract interface. An interface block with a generic specification is a generic interface block. An interface block with neither ABSTRACT nor a generic specification is a specific interface block.

3 The name of the entity declared by an interface body is the function-name in the function-stmt or subroutine-name in the subroutine-stmt that begins the interface body.

4 A module procedure interface body is an interface body whose initial statement contains the keyword MODULE. It specifies the interface for a separate module procedure (15.6.2.5). A separate module procedure is accessible by use association if and only if its interface body is declared in the specification part of a module and is public. If a corresponding (15.6.2.5) separate module procedure is not defined, the interface may be used to specify an explicit specific interface but the procedure shall not be used in any other way.

5 An interface body in a generic or specific interface block specifies the EXTERNAL attribute and an explicit specific interface for an external procedure, dummy procedure, or procedure pointer. If the name of the declared procedure is that of a dummy argument in the subprogram containing the interface body, the procedure is a dummy procedure. If the procedure has the POINTER attribute, it is a procedure pointer. If it is not a dummy procedure or procedure pointer, it is an external procedure.

An interface body specifies all of the characteristics of the explicit specific interface or abstract interface. The specification part of an interface body may specify attributes or define values for data entities that do not determine characteristics of the procedure. Such specifications have no effect.

7 If an explicit specific interface for an external procedure is specified by an interface body or a procedure declaration statement (15.4.3.6), the characteristics shall be consistent with those specified in the procedure definition, except that the interface may specify a procedure that is not pure even if the procedure is defined to be pure. An interface for a procedure defined by an ENTRY statement may be specified by using the entry name as the procedure name in the interface body. If an external procedure does not exist in the program, an interface body for it may be used to specify an explicit specific interface but the procedure shall not be used in any other way. A procedure shall not have more than one explicit specific interface in a given scoping unit, except that if the interface is accessed by use association, there may be more than one local name for the procedure. If a procedure is accessed by use association, each access shall be to the same procedure declaration or definition.

NOTE 15.3
The dummy argument names in an interface body can be different from the corresponding dummy argument names in the procedure definition because the name of a dummy argument is not a characteristic.
NOTE 15.4

An example of a specific interface block is:

```fortran
INTERFACE
  SUBROUTINE EXT1 (X, Y, Z)
    REAL, DIMENSION (100, 100) :: X, Y, Z
  END SUBROUTINE EXT1
  SUBROUTINE EXT2 (X, Z)
    REAL X
    COMPLEX (KIND = 4) Z (2000)
  END SUBROUTINE EXT2
  FUNCTION EXT3 (P, Q)
    LOGICAL EXT3
    INTEGER P (1000)
    LOGICAL Q (1000)
  END FUNCTION EXT3
END INTERFACE
```

This interface block specifies explicit interfaces for the three external procedures EXT1, EXT2, and EXT3. Invocations of these procedures can use argument keywords (15.5.2); for example:

```fortran
PRINT *, EXT3 (Q = P_MASK (N+1 : N+1000), P = ACTUAL_P)
```

15.4.3.3 GENERIC statement

A GENERIC statement specifies a generic identifier for one or more specific procedures, in the same way as a generic interface block that does not contain interface bodies.

R1510 generic-stmt is GENERIC [ , access-spec ] :: generic-spec => specific-procedure-list

C1510 (R1510) A specific-procedure in a GENERIC statement shall not specify a procedure that was specified previously in any accessible interface with the same generic identifier.

If access-spec appears, it specifies the accessibility (8.5.2) of generic-spec.

15.4.3.4 Generic interfaces

15.4.3.4.1 Generic identifiers

A generic interface block specifies a generic interface for each of the procedures in the interface block. The PROCEDURE statement lists procedure pointers, external procedures, dummy procedures, or module procedures that have this generic interface. A GENERIC statement specifies a generic interface for each of the procedures named in its specific-procedure-list. A generic interface is always explicit.

The generic-spec in an interface-stmt is a generic identifier for all the procedures in the interface block. The generic-spec in a GENERIC statement is a generic identifier for all of the procedures named in its specific-procedure-list. The rules specifying how any two procedures with the same generic identifier shall differ are given in 15.4.3.4.5. They ensure that any generic invocation applies to at most one specific procedure. If a specific procedure in a generic interface has a function dummy argument, that argument shall have its type and type parameters explicitly declared in the specific interface.

A generic name is a generic identifier that refers to all of the procedure names in the interface block. A generic name may be the same as any one of the procedure names in the interface block, or the same as any accessible generic name.

A generic name may be the same as a derived-type name, in which case all of the procedures in the interface block shall be functions.
An interface-stmt having a defined-io-generic-spec is an interface for a defined input/output procedure (12.6.4.8).

NOTE 15.5

An example of a generic procedure interface is:

```
INTERFACE SWITCH

SUBROUTINE INT_SWITCH (X, Y)
    INTEGER, INTENT (INOUT) :: X, Y
END SUBROUTINE INT_SWITCH

SUBROUTINE REAL_SWITCH (X, Y)
    REAL, INTENT (INOUT) :: X, Y
END SUBROUTINE REAL_SWITCH

SUBROUTINE COMPLEX_SWITCH (X, Y)
    COMPLEX, INTENT (INOUT) :: X, Y
END SUBROUTINE COMPLEX_SWITCH

END INTERFACE SWITCH
```

Any of these three subroutines (INT_SWITCH, REAL_SWITCH, COMPLEX_SWITCH) can be referenced with the generic name SWITCH, as well as by its specific name. For example, a reference to INT_SWITCH could take the form:

```
CALL SWITCH (MAX_VAL, LOC_VAL) ! MAX_VAL and LOC_VAL are of type INTEGER
```

NOTE 15.6

A type-bound-generic-stmt within a derived-type definition (7.5.5) specifies a generic identifier for a set of type-bound procedures.

15.4.3.4.2 Defined operations

1 If OPERATOR is specified in a generic specification, all of the procedures specified in the generic interface shall be functions that may be referenced as defined operations (10.1.6, 15.5). In the case of functions of two arguments, infix binary operator notation is implied. In the case of functions of one argument, prefix operator notation is implied. OPERATOR shall not be specified for functions with no arguments or for functions with more than two arguments. The dummy arguments shall be nonoptional dummy data objects and shall have the INTENT (IN) or VALUE attribute. The function result shall not have assumed character length. If the operator is an intrinsic-operator (R608), the number of dummy arguments shall be consistent with the intrinsic uses of that operator, and the types, kind type parameters, or ranks of the dummy arguments shall differ from those required for the intrinsic operation (10.1.5).

2 A defined operation is treated as a reference to the function. For a unary defined operation, the operand corresponds to the function’s dummy argument; for a binary operation, the left-hand operand corresponds to the first dummy argument of the function and the right-hand operand corresponds to the second dummy argument. All restrictions and constraints that apply to actual arguments in a reference to the function also apply to the corresponding operands in the expression as if they were used as actual arguments.

3 A given defined operator may, as with generic names, apply to more than one function, in which case it is generic in exact analogy to generic procedure names. For intrinsic operator symbols, the generic properties include the intrinsic operations they represent. Because both forms of each relational operator have the same interpretation (10.1.6.2), extending one form (such as <) has the effect of defining both forms (<= and .LE.).

NOTE 15.7

An example of the use of the OPERATOR generic specification is:

```
INTERFACE OPERATOR (*)

FUNCTION BOOLEAN_AND (B1, B2)
```

ISO/IEC JTC 1/SC 22/WG5/N2118 307
NOTE 15.7 (cont.)

LOGICAL, INTENT (IN) :: B1 (:), B2 (SIZE (B1))
LOGICAL :: BOOLEAN_AND (SIZE (B1))
END FUNCTION BOOLEAN_AND
END INTERFACE OPERATOR ( * )

This allows, for example

SENSOR (1:N) * ACTION (1:N)

as an alternative to the function call

BOOLEAN_AND (SENSOR (1:N), ACTION (1:N)) ! SENSOR and ACTION are
! of type LOGICAL

15.4.3.4.3 Defined assignments

If ASSIGNMENT ( = ) is specified in a generic specification, all the procedures in the generic interface shall
be subroutines that may be referenced as defined assignments (10.2.1.4, 10.2.1.5). Defined assignment may, as
with generic names, apply to more than one subroutine, in which case it is generic in exact analogy to generic
procedure names.

Each of these subroutines shall have exactly two dummy arguments. The dummy arguments shall be nonoptional
dummy data objects. The first argument shall have INTENT (OUT) or INTENT (INOUT) and the second
argument shall have the INTENT (IN) or VALUE attribute. Either the second argument shall be an array whose
rank differs from that of the first argument, the declared types and kind type parameters of the arguments shall
not conform as specified in Table 10.8, or the first argument shall be of derived type. A defined assignment is
treated as a reference to the subroutine, with the left-hand side as the first argument and the right-hand side
enclosed in parentheses as the second argument. All restrictions and constraints that apply to actual arguments
in a reference to the subroutine also apply to the left-hand-side and to the right-hand-side enclosed in parentheses
as if they were used as actual arguments. The ASSIGNMENT generic specification specifies that assignment is
extended or redefined.

NOTE 15.8

An example of the use of the ASSIGNMENT generic specification is:

INTERFACE ASSIGNMENT ( = )

SUBROUTINE LOGICAL_TO_NUMERIC (N, B)
  INTEGER, INTENT (OUT) :: N
  LOGICAL, INTENT (IN) :: B
END SUBROUTINE LOGICAL_TO_NUMERIC

SUBROUTINE CHAR_TO_STRING (S, C)
  USE STRING_MODULE ! Contains definition of type STRING
  TYPE (STRING), INTENT (OUT) :: S ! A variable-length string
  CHARACTER (*), INTENT (IN) :: C
END SUBROUTINE CHAR_TO_STRING
END INTERFACE ASSIGNMENT ( = )

Example assignments are:

KOUNT = SENSOR (J) ! CALL LOGICAL_TO_NUMERIC (KOUNT, (SENSOR (J)))
NOTE = ’89AB’ ! CALL CHAR_TO_STRING (NOTE, (’89AB’))
NOTE 15.9
A procedure whose second dummy argument has the ALLOCATABLE or POINTER attribute cannot be accessed via defined assignment, even if it given the ASSIGNMENT ( = ) generic identifier. This is because the actual argument associated with that dummy argument is the right-hand side of the assignment enclosed in parentheses, which makes the actual argument an expression that does not have the ALLOCATABLE, POINTER, or TARGET attribute.

15.4.3.4.4 Defined input/output procedure interfaces

1 All of the procedures specified in an interface block for a defined input/output procedure shall be subroutines that have interfaces as described in 12.6.4.8.2.

15.4.3.4.5 Restrictions on generic declarations

1 This subclause contains the rules that shall be satisfied by every pair of specific procedures that have the same generic identifier within the scope of the identifier. If a generic procedure is accessed from a module, the rules apply to all the specific versions even if some of them are inaccessible by their specific names.

NOTE 15.10
In most scoping units, the possible sources of procedures with a particular generic identifier are the accessible interface blocks and the generic bindings other than names for the accessible objects in that scoping unit. In a type definition, they are the generic bindings, including those from a parent type.

2 A dummy argument is type, kind, and rank compatible, or TKR compatible, with another dummy argument if the first is type compatible with the second, the kind type parameters of the first have the same values as the corresponding kind type parameters of the second, and both have the same rank or either is assumed-rank.

3 Two dummy arguments are distinguishable if

- one is a procedure and the other is a data object,
- they are both data objects or known to be functions, and neither is TKR compatible with the other,
- one has the ALLOCATABLE attribute and the other has the POINTER attribute and not the INTENT (IN) attribute, or
- one is a function with nonzero rank and the other is not known to be a function.

C1511 Within the scope of a generic operator, if two procedures with that identifier have the same number of arguments, one shall have a dummy argument that corresponds by position in the argument list to a dummy argument of the other that is distinguishable from it.

C1512 Within the scope of the generic ASSIGNMENT (=) identifier, if two procedures have that identifier, one shall have a dummy argument that corresponds by position in the argument list to a dummy argument of the other that is distinguishable from it.

C1513 Within the scope of a defined-io-generic-spec, if two procedures have that generic identifier, their dtv arguments (12.6.4.8.2) shall be distinguishable.

C1514 Within the scope of a generic name, each pair of procedures identified by that name shall both be subroutines or both be functions, and

(1) there is a non-passed-object dummy data object in one or the other of them such that

(a) the number of dummy data objects in one that are nonoptional, are not passed-object, and with which that dummy data object is TKR compatible, possibly including that dummy data object itself,

exceeds

(b) the number of non-passed-object dummy data objects, both optional and nonoptional, in the other that are not distinguishable from that dummy data object,
(2) the number of nonoptional dummy procedures in one of them exceeds the number of dummy procedures in the other,
(3) both have passed-object dummy arguments and the passed-object dummy arguments are distinguishable, or
(4) at least one of them shall have both
    (a) a nonoptional non-passed-object dummy argument at an effective position such that either
        the other procedure has no dummy argument at that effective position or the dummy argument at that position is distinguishable from it, and
    (b) a nonoptional non-passed-object dummy argument whose name is such that either the other procedure has no dummy argument with that name or the dummy argument with that name is distinguishable from it,
    and the dummy argument that disambiguates by position shall either be the same as or occur earlier in the argument list than the one that disambiguates by name.

4 The effective position of a dummy argument is its position in the argument list after any passed-object dummy argument has been removed.

5 Within the scope of a generic name that is the same as the generic name of an intrinsic procedure, the intrinsic procedure is not accessible by its generic name if the procedures in the interface and the intrinsic procedure are not all functions or not all subroutines. If a generic invocation is consistent with both a specific procedure from an interface and an accessible intrinsic procedure, it is the specific procedure from the interface that is referenced.

NOTE 15.11 An extensive explanation of the application of these rules is in C.10.6.

15.4.3.5 EXTERNAL statement

1 An EXTERNAL statement specifies the EXTERNAL attribute (8.5.9) for a list of names.

R1511 external-stmt is EXTERNAL [ :: ] external-name-list

2 The appearance of the name of a block data program unit in an EXTERNAL statement confirms that the block data program unit is a part of the program.

NOTE 15.12 For explanatory information on potential portability problems with external procedures, see subclause C.10.1.

NOTE 15.13 An example of an EXTERNAL statement is:

EXTERNAL FOCUS

15.4.3.6 Procedure declaration statement

1 A procedure declaration statement declares procedure pointers, dummy procedures, and external procedures. It specifies the EXTERNAL attribute (8.5.9) for all entities in the proc-decl-list.

R1512 procedure-declaration-stmt is PROCEDURE ( [ proc-interface ] ) ::

■ [ [ , proc-attr-spec ] ... :: ] proc-decl-list

R1513 proc-interface is interface-name or declaration-type-spec
R1514 proc-attr-spec is access-spec
or proc-language-binding-spec
or INTENT ( intent-spec )
or OPTIONAL
or POINTER
or PROTECTED
or SAVE

R1515 proc-decl is procedure-entity-name [ => proc-pointer-init ]

R1516 interface-name is name

R1517 proc-pointer-init is null-init
or initial-proc-target

R1518 initial-proc-target is procedure-name

C1515 (R1516) The name shall be the name of an abstract interface or of a procedure that has an explicit interface. If name is declared by a procedure-declaration-stmt it shall be previously declared. If name denotes an intrinsic procedure it shall be one that is listed in Table 16.2.

C1516 (R1516) The name shall not be the same as a keyword that specifies an intrinsic type.

C1517 (R1512) If a proc-interface describes an elemental procedure, each procedure-entity-name shall specify an external procedure.

C1518 (R1515) If => appears in proc-decl, the procedure entity shall have the POINTER attribute.

C1519 (R1518) The procedure-name shall be the name of a nonelemental external or module procedure, or a specific intrinsic function listed in Table 16.2.

C1520 (R1512) If proc-language-binding-spec with NAME= is specified, then proc-decl-list shall contain exactly one proc-decl, which shall neither have the POINTER attribute nor be a dummy procedure.

C1521 (R1512) If proc-language-binding-spec is specified, the proc-interface shall appear, it shall be an interface-name, and interface-name shall be declared with a proc-language-binding-spec.

2 If proc-interface appears and consists of interface-name, it specifies an explicit specific interface (15.4.3.2) for the declared procedure entities. The abstract interface (15.4) is that specified by the interface named by interface-name. The interface specified by interface-name shall not depend on any characteristic of a procedure identified by a procedure-entity-name in the proc-decl-list of the same procedure declaration statement.

3 If proc-interface appears and consists of declaration-type-spec, it specifies that the declared procedure entities are functions having implicit interfaces and the specified result type. If a type is specified for an external function, its function definition (15.6.2.2) shall specify the same result type and type parameters.

4 If proc-interface does not appear, the procedure declaration statement does not specify whether the declared procedure entities are subroutines or functions.

5 If a proc-attr-spec other than a proc-language-binding-spec appears, it specifies that the declared procedure entities have that attribute. These attributes are described in 8.5. If a proc-language-binding-spec with NAME= appears, it specifies a binding label or its absence, as described in 18.10.2. A proc-language-binding-spec without NAME= is allowed, but is redundant with the proc-interface required by C1521.

6 If => appears in a proc-decl in a procedure-declaration-stmt it specifies the initial association status of the corresponding procedure entity, and implies the SAVE attribute, which may be confirmed by explicit specification. If => null-init appears, the procedure entity is initially disassociated. If => initial-proc-target appears, the procedure entity is initially associated with the target.
If `procedure-entity-name` has an explicit interface, its characteristics shall be the same as `initial-proc-target` except that `initial-proc-target` may be pure even if `procedure-entity-name` is not pure and `initial-proc-target` may be an elemental intrinsic procedure.

If the characteristics of `procedure-entity-name` or `initial-proc-target` are such that an explicit interface is required, both `procedure-entity-name` and `initial-proc-target` shall have an explicit interface.

If `procedure-entity-name` has an implicit interface and is explicitly typed or referenced as a function, `initial-proc-target` shall be a function. If `procedure-entity-name` has an implicit interface and is referenced as a subroutine, `initial-proc-target` shall be a subroutine.

If `procedure-entity-name` has an implicit interface and is explicitly typed or referenced as a function, `initial-proc-target` shall be a function. If `procedure-entity-name` has an implicit interface and is referenced as a subroutine, `initial-proc-target` shall be a subroutine.

If `initial-proc-target` and `procedure-entity-name` are functions, their results shall have the same characteristics.

**NOTE 15.14**

The following code illustrates procedure declaration statements. Note 10.47 illustrates the use of the P and BESSEL defined by this code.

```
ABSTRACT INTERFACE
  FUNCTION REAL_FUNC (X)
    REAL, INTENT (IN) :: X
    REAL :: REAL_FUNC
  END FUNCTION REAL_FUNC
END INTERFACE

INTERFACE
  SUBROUTINE SUB (X)
    REAL, INTENT (IN) :: X
  END SUBROUTINE SUB
END INTERFACE

!-- Some external or dummy procedures with explicit interface.
PROCEDURE (REAL_FUNC) :: BESSEL, GFUN
PROCEDURE (SUB) :: PRINT_REAL
!-- Some procedure pointers with explicit interface,
!-- one initialized to NULL().
PROCEDURE (REAL_FUNC), POINTER :: P, R => NULL ()
PROCEDURE (REAL_FUNC), POINTER :: PTR_TO_GFUN
!-- A derived type with a procedure pointer component ...
  TYPE STRUCT_TYPE
    PROCEDURE (REAL_FUNC), POINTER, NOPASS :: COMPONENT
  END TYPE STRUCT_TYPE
!-- ... and a variable of that type.
  TYPE(STRUCT_TYPE) :: STRUCT
!-- An external or dummy function with implicit interface
PROCEDURE (REAL) :: PSI
```

**15.4.3.7 INTRINSIC statement**

An INTRINSIC statement specifies the INTRINSIC attribute (8.5.11) for a list of names.

| R1519 | intrinsic-stmt is INTRINSIC [ :: ] intrinsic-procedure-name-list |
| C1522 | (R1519) Each intrinsic-procedure-name shall be the name of an intrinsic procedure. |
15.4.3.8  Implicit interface specification

1 If the interface of a function is implicit, the type and type parameters of the function result are specified by an implicit or explicit type specification of the function name. The type, type parameters, and shape of the dummy arguments of a procedure invoked from where the interface of the procedure is implicit shall be such that each actual argument is consistent with the characteristics of the corresponding dummy argument.

15.5  Procedure reference

15.5.1  Syntax of a procedure reference

1 The form of a procedure reference is dependent on the interface of the procedure or procedure pointer, but is independent of the means by which the procedure is defined. The forms of procedure references are as follows.

R1520   function-reference          is   procedure-designator ( [ actual-arg-spec-list ] )
C1523   (R1520) The procedure-designator shall designate a function.
C1524   (R1520) The actual-arg-spec-list shall not contain an alt-return-spec.
R1521   call-stmt          is   CALL procedure-designator [ ( [ actual-arg-spec-list ] ) ]
C1525   (R1521) The procedure-designator shall designate a subroutine.
R1522   procedure-designator          is   procedure-name
       or   proc-component-ref
       or   data-ref % binding-name
C1526   (R1522) A procedure-name shall be a generic name or the name of a procedure.
C1527   (R1522) A binding-name shall be a binding name (7.5.5) of the declared type of data-ref.
C1528   (R1522) A data-ref shall not be a polymorphic subobject of a coindexed object.
C1529   (R1522) If data-ref is an array, the referenced type-bound procedure shall have the PASS attribute.

2 The data-ref in a procedure-designator shall not be an unallocated allocatable variable or a pointer that is not associated.

3 Resolving references to type-bound procedures is described in 15.5.6.

4 A function may also be referenced as a defined operation (10.1.6). A subroutine may also be referenced as a defined assignment (10.2.1.4, 10.2.1.5), by defined input/output (12.6.4.8), or by finalization (7.5.6).

NOTE 15.15
When resolving type-bound procedure references, constraints on the use of coindexed objects ensure that the coindexed object (on the remote image) has the same dynamic type as the corresponding object on the local image. Thus a processor can resolve the type-bound procedure using the coarray variable on its own image and pass the coindexed object as the actual argument.

R1523   actual-arg-spec          is   [ keyword = ] actual-arg
R1524   actual-arg          is   expr
       or   variable
       or   procedure-name
       or   proc-component-ref
       or   alt-return-spec
R1525  alt-return-spec is * label

C1530  (R1523) The keyword = shall not appear if the interface of the procedure is implicit.
C1531  (R1523) The keyword = shall not be omitted from an actual-arg-spec unless it has been omitted from each preceding actual-arg-spec in the argument list.
C1532  (R1523) Each keyword shall be the name of a dummy argument in the explicit interface of the procedure.
C1533  (R1524) A nonintrinsic elemental procedure shall not be used as an actual argument.
C1534  (R1524) A procedure-name shall be the name of an external, internal, module, or dummy procedure, a specific intrinsic function listed in Table 16.2, or a procedure pointer.
C1535  (R1524) expr shall not be a variable.
C1536  (R1525) The label shall be the statement label of a branch target statement that appears in the same inclusive scope as the call-stmt.
C1537  An actual argument that is a coindexed object shall not have a pointer ultimate component.

NOTE 15.16
Examples of procedure reference using procedure pointers:

P => BESSEL
WRITE (*, *) P(2.5) !-- BESSEL(2.5)

S => PRINT_REAL
CALL S(3.14)

NOTE 15.17
An internal procedure cannot be invoked using a procedure pointer from either Fortran or C after the host instance completes execution, because the pointer is then undefined. While the host instance is active, however, if an internal procedure was passed as an actual argument or is the target of a procedure pointer, it could be invoked from outside of the host subprogram.

Assume there is a procedure with the following interface that calculates \( \int_a^b f(x) \, dx \).

INTERFACE
FUNCTION INTEGRATE(F, A, B) RESULT(INTEGRAL) BIND(C)
  USE ISO_C_BINDING
  INTERFACE
    FUNCTION F(X) BIND(C) ! Integrand
      USE ISO_C_BINDING
      REAL(C_FLOAT), VALUE :: X
      REAL(C_FLOAT) :: F
    END FUNCTION
  END INTERFACE
  REAL(C_FLOAT), VALUE :: A, B ! Bounds
  REAL(C_FLOAT) :: INTEGRAL
END FUNCTION INTEGRATE
END INTERFACE

This procedure can be called from Fortran or C, and could be written in either Fortran or C. The argument F representing the mathematical function \( f(x) \) can be written as an internal procedure; this internal procedure will have access to any host instance local variables necessary to actually calculate \( f(x) \). For example:
NOTE 15.17 (cont.)

REAL FUNCTION MY_INTEGRATION(N, A, B) RESULT(INTEGRAL)
    ! Integrate f(x)=x^n over [a,b]
USE ISO_C_BINDING
INTEGER, INTENT(IN) :: N
REAL, INTENT(IN) :: A, B

INTEGRAL = INTEGRATE(MY_F, REAL (A, C_FLOAT), REAL (B, C_FLOAT))
    ! This will call the internal function MY_F to calculate f(x).
    ! The above interface of INTEGRATE needs to be explicit and available.
CONTAINS

REAL(C_FLOAT) FUNCTION MY_F(X) BIND(C) ! Integrand
REAL(C_FLOAT), VALUE :: X
MY_F = X**N ! N is taken from the host instance of MY_INTEGRATION.
END FUNCTION

END FUNCTION MY_INTEGRATION

The function INTEGRATE cannot retain a function pointer to MY_F and use it after INTEGRATE has finished execution, because the host instance of MY_F might no longer exist, making the pointer undefined. If such a pointer is retained, then it can only be used to invoke MY_F during the execution of the host instance of MY_INTEGRATION called from INTEGRATE.

15.5.2 Actual arguments, dummy arguments, and argument association

15.5.2.1 Argument correspondence

In either a subroutine reference or a function reference, the actual argument list identifies the correspondence between the actual arguments and the dummy arguments of the procedure. This correspondence may be established either by keyword or by position. If an argument keyword appears, the actual argument corresponds to the dummy argument whose name is the same as the argument keyword (using the dummy argument names from the interface accessible by the procedure reference). In the absence of an argument keyword, an actual argument corresponds to the dummy argument occupying the corresponding position in the reduced dummy argument list; that is, the first actual argument corresponds to the first dummy argument in the reduced list, the second actual argument corresponds to the second dummy argument in the reduced list, etc. The reduced dummy argument list is either the full dummy argument list or, if there is a passed-object dummy argument (7.5.4.5), the dummy argument list with the passed-object dummy argument omitted. Exactly one actual argument shall correspond to each nonoptional dummy argument. At most one actual argument shall correspond to each optional dummy argument. Each actual argument shall correspond to a dummy argument.

NOTE 15.18

For example, the procedure defined by

SUBROUTINE SOLVE (FUNCT, SOLUTION, METHOD, STRATEGY, PRINT)
    INTERFACE
        FUNCTION FUNCT (X)
            REAL FUNCT, X
        END FUNCTION FUNCT
    END INTERFACE
    REAL SOLUTION
    INTEGER, OPTIONAL :: METHOD, STRATEGY, PRINT
    ...
NOTE 15.18 (cont.)

```fortran
CALL SOLVE (FUN, SOL, PRINT = 6)
```

provided its interface is explicit, and if the interface is specified by an interface body, the name of the last argument is PRINT.

15.5.2.2 The passed-object dummy argument and argument correspondence

1 In a reference to a type-bound procedure, or a procedure pointer component, that has a passed-object dummy argument (7.5.4.5), the data-ref of the function-reference or call-stmt corresponds, as an actual argument, with the passed-object dummy argument.

15.5.2.3 Argument association

2 Except in references to intrinsic inquiry functions, a pointer actual argument that corresponds to a nonoptional nonpointer dummy argument shall be pointer associated with a target.

3 If a nonpointer dummy argument without the VALUE attribute corresponds to a pointer actual argument that is pointer associated with a target,
   • if the dummy argument is polymorphic, it becomes argument associated with that target;
   • if the dummy argument is nonpolymorphic, it becomes argument associated with the declared type part of that target.

4 If a present nonpointer dummy argument without the VALUE attribute corresponds to a nonpointer actual argument,
   • if the dummy argument is polymorphic, it becomes argument associated with that actual argument;
   • if the dummy argument is nonpolymorphic, it becomes argument associated with the declared type part of that actual argument.

5 A present dummy argument with the VALUE attribute becomes argument associated with a definable anonymous data object whose initial value is the value of the actual argument.

6 A present pointer dummy argument that corresponds to a pointer actual argument becomes argument associated with that actual argument. A present pointer dummy argument that does not correspond to a pointer actual argument is not argument associated.

7 The entity that is argument associated with a dummy argument is called its effective argument.

8 The ultimate argument is the effective argument if the effective argument is not a dummy argument or a subobject of a dummy argument. If the effective argument is a dummy argument, the ultimate argument is the ultimate argument of that dummy argument. If the effective argument is a subobject of a dummy argument, the ultimate argument is the corresponding subobject of the ultimate argument of that dummy argument.

NOTE 15.19

For the sequence of subroutine calls

```fortran
INTEGER :: X(100)
CALL SUBA (X)
...
SUBROUTINE SUBA(A)
INTEGER :: A(:)
CALL SUBB (A(1:5), A(5:1:-1))
...
SUBROUTINE SUBB(B, C)
```
NOTE 15.19 (cont.)

```
INTEGER :: B(:), C(:)
```

the ultimate argument of B is X(1:5). The ultimate argument of C is X(5:1:-1) and this is not the same object as the ultimate argument of B.

NOTE 15.20

Fortran argument association is usually similar to call by reference and call by value-result. If the VALUE attribute is specified, the effect is as if the actual argument is assigned to a temporary, and the temporary is then argument associated with the dummy argument. Subsequent changes to the value or definition status of the dummy argument do not affect the actual argument. The actual mechanism by which this happens is determined by the processor.

15.5.2.4 Ordinary dummy variables

1. The requirements in this subclause apply to actual arguments that correspond to nonallocatable nonpointer dummy data objects.

2. The dummy argument shall be type compatible with the actual argument. If the actual argument is a polymorphic coindexed object, the dummy argument shall not be polymorphic. If the actual argument is a polymorphic assumed-size array, the dummy argument shall be polymorphic. If the actual argument is of a derived type that has type parameters, type-bound procedures, or final subroutines, the dummy argument shall not be assumed-type.

3. The kind type parameter values of the actual argument shall agree with the corresponding ones of the dummy argument. The length type parameter values of a present actual argument shall agree with the corresponding ones of the dummy argument that are not assumed, except for the case of the character length parameter of an actual argument of type character with default kind or C character kind (18.2.2) associated with a dummy argument that is not assumed-shape or assumed-rank.

4. If a present scalar dummy argument is of type character with default kind or C character kind, the length len of the dummy argument shall be less than or equal to the length of the actual argument. The dummy argument becomes associated with the leftmost len characters of the actual argument. If a present array dummy argument is of type character with default kind or C character kind and is not assumed-shape or assumed-rank, it becomes associated with the leftmost characters of the actual argument element sequence (15.5.2.11).

5. The values of assumed type parameters of a dummy argument are assumed from the corresponding type parameters of its effective argument.

6. If the actual argument is a coindexed object with an allocatable ultimate component, the dummy argument shall have the INTENT (IN) or the VALUE attribute.

NOTE 15.21

If the actual argument is a coindexed object, a processor that uses distributed memory might create a copy on the executing image of the actual argument, including copies of any allocated allocatable subobjects, and associate the dummy argument with that copy. If necessary, on return from the procedure, the value of the copy would be copied back to the actual argument.

7. Except in references to intrinsic inquiry functions, if the dummy argument is nonoptional and the actual argument is allocatable, the corresponding actual argument shall be allocated.

8. If the dummy argument does not have the TARGET attribute, any pointers associated with the effective argument do not become associated with the corresponding dummy argument on invocation of the procedure. If such a dummy argument is used as an actual argument that corresponds to a dummy argument with the TARGET attribute, whether any pointers associated with the original effective argument become associated with the dummy argument with the TARGET attribute is processor dependent.
9. If the dummy argument has the **TARGET** attribute, does not have the **VALUE** attribute, and either the effective argument is simply contiguous or the dummy argument is scalar, assumed-rank, or assumed-shape, and does not have the CONTIGUOUS attribute, and the effective argument has the **TARGET** attribute but is not a coindexed object or an array section with a vector subscript then

- any pointers associated with the effective argument become associated with the corresponding dummy argument on invocation of the procedure, and
- when execution of the procedure completes, any pointers that do not become undefined (19.5.2.5) and are associated with the dummy argument remain associated with the effective argument.

10. If the dummy argument has the **TARGET** attribute and is an explicit-shape array, an assumed-shape array with the CONTIGUOUS attribute, an assumed-rank object with the CONTIGUOUS attribute, or an assumed-size array, and the effective argument has the **TARGET** attribute but is not simply contiguous and is not an array section with a vector subscript then

- on invocation of the procedure, whether any pointers associated with the effective argument become associated with the corresponding dummy argument is processor dependent, and
- when execution of the procedure completes, the pointer association status of any pointer that is pointer associated with the dummy argument is processor dependent.

11. If the dummy argument has the **TARGET** attribute and the effective argument does not have the **TARGET** attribute or is an array section with a vector subscript, any pointers associated with the dummy argument become undefined when execution of the procedure completes.

12. If the dummy argument has the **TARGET** attribute and the **VALUE** attribute, any pointers associated with the dummy argument become undefined when execution of the procedure completes.

13. If the actual argument is a coindexed scalar, the corresponding dummy argument shall be scalar. If the actual argument is a noncoindexed scalar, the corresponding dummy argument shall be scalar unless the actual argument is default character, of type character with the C character kind (18.2.2), or is an element or substring of an element of an array that is not an assumed-shape, pointer, or polymorphic array. If the procedure is nonelemental and is referenced by a generic name or as a defined operator or defined assignment, the ranks of the actual arguments and corresponding dummy arguments shall agree.

14. If a dummy argument is an assumed-shape array, the rank of the actual argument shall be the same as the rank of the dummy argument; the actual argument shall not be an assumed-size array (including an array element designator or an array element substring designator).

15. An actual argument of any rank may correspond to an assumed-rank dummy argument. The rank and shape of the dummy argument are the rank and shape of the corresponding actual argument. If the rank is nonzero, the lower and upper bounds of the dummy argument are those that would be given by the intrinsic functions **LBOUND** and **UBOUND** respectively if applied to the actual argument, except that when the actual argument is assumed-size, the upper bound of the last dimension of the dummy argument is 2 less than the lower bound of that dimension.

16. Except when a procedure reference is elemental (15.8), each element of an array actual argument or of a sequence in a sequence association (15.5.2.11) is associated with the element of the dummy array that has the same position in array element order (9.5.3.2).

**NOTE 15.22**

For default character sequence associations, the interpretation of element is provided in 15.5.2.11.

17. A scalar dummy argument of a nonelemental procedure shall correspond only to a scalar actual argument.

18. If a dummy argument has **INTENT** (OUT) or **INTENT** (INOUT), the actual argument shall be definable. If a dummy argument has **INTENT** (OUT), the effective argument becomes undefined at the time the association is
established, except for direct components of an object of derived type for which default initialization has been specified.

If the procedure is nonelemental, the dummy argument does not have the VALUE attribute, and the actual argument is an array section having a vector subscript, the dummy argument is not definable and shall not have the ASYNCHRONOUS, INTENT (OUT), INTENT (INOUT), or VOLATILE attributes.

If the dummy argument has a coarray ultimate component, the corresponding actual argument shall have the VOLATILE attribute if and only if the dummy argument has the VOLATILE attribute.

NOTE 15.23
Argument intent specifications serve several purposes. See Note 8.17.

NOTE 15.24
For more explanatory information on targets as dummy arguments, see subclause C.10.4.

C1538 An actual argument that is a coindexed object with the ASYNCHRONOUS or VOLATILE attribute shall not correspond to a dummy argument that has either the ASYNCHRONOUS or VOLATILE attribute, unless the dummy argument has the VALUE attribute.

C1539 (R1524) If an actual argument is a nonpointer array that has the ASYNCHRONOUS or VOLATILE attribute but is not simply contiguous (9.5.4), and the corresponding dummy argument has either the ASYNCHRONOUS or VOLATILE attribute, but does not have the VALUE attribute, that dummy argument shall be assumed-shape or assumed-rank and shall not have the CONTIGUOUS attribute.

C1540 (R1524) If an actual argument is an array pointer that has the ASYNCHRONOUS or VOLATILE attribute but does not have the CONTIGUOUS attribute, and the corresponding dummy argument has either the ASYNCHRONOUS or VOLATILE attribute, but does not have the VALUE attribute, that dummy argument shall be an array pointer, an assumed-shape array without the CONTIGUOUS attribute, or an assumed-rank entity without the CONTIGUOUS attribute.

NOTE 15.25
The constraints on an actual argument with the ASYNCHRONOUS or VOLATILE attribute that corresponds to a dummy argument with either the ASYNCHRONOUS or VOLATILE attribute are designed to avoid forcing a processor to use the so-called copy-in/copy-out argument passing mechanism. Making a copy of an actual argument whose value is likely to change due to an asynchronous input/output operation completing or in some unpredictable manner will cause the new value to be lost when a called procedure returns and the copy-out overwrites the actual argument.

15.5.2.5 Allocatable and pointer dummy variables

The requirements in this subclause apply to an actual argument with the ALLOCATABLE or POINTER attribute that corresponds to a dummy argument with the same attribute.

The actual argument shall be polymorphic if and only if the associated dummy argument is polymorphic, and either both the actual and dummy arguments shall be unlimited polymorphic, or the declared type of the actual argument shall be the same as the declared type of the dummy argument.

NOTE 15.26
The dynamic type of a polymorphic allocatable or pointer dummy argument can change as a result of execution of an ALLOCATE statement or pointer assignment in the subprogram. Because of this the corresponding actual argument needs to be polymorphic and have a declared type that is the same as the declared type of the dummy argument or an extension of that type. However, type compatibility requires that the declared type of the dummy argument be the same as, or an extension of, the type of the actual argument. Therefore, the dummy and actual arguments need to have the same declared type.
NOTE 15.26 (cont.)

Dynamic type information is not maintained for a nonpolymorphic allocatable or pointer dummy argument. However, allocating or pointer-assigning such a dummy argument would require maintenance of this information if the corresponding actual argument is polymorphic. Therefore, the corresponding actual argument needs to be nonpolymorphic.

3 The rank of the actual argument shall be the same as that of the dummy argument, unless the dummy argument is assumed-rank. The type parameter values of the actual argument shall agree with the corresponding ones of the dummy argument that are not assumed or deferred. The values of assumed type parameters of the dummy argument are assumed from the corresponding type parameters of its effective argument.

4 The actual argument shall have deferred the same type parameters as the dummy argument.

5 If the actual argument is a coindexed object, the dummy argument shall have the INTENT (IN) attribute.

15.5.2.6 Allocatable dummy variables

1 The requirements in this subclause apply to actual arguments that correspond to allocatable dummy data objects.

2 The actual argument shall be allocatable. It is permissible for the actual argument to have an allocation status of unallocated.

3 The corank of the actual argument shall be the same as that of the dummy argument.

4 If the dummy argument does not have the TARGET attribute, any pointers associated with the actual argument do not become associated with the corresponding dummy argument on invocation of the procedure. If such a dummy argument is used as an actual argument that is associated with a dummy argument with the TARGET attribute, whether any pointers associated with the original actual argument become associated with the dummy argument with the TARGET attribute is processor dependent.

5 If the dummy argument has the TARGET attribute, does not have the INTENT (OUT) or VALUE attribute, and the corresponding actual argument has the TARGET attribute then

- any pointers associated with the actual argument become associated with the corresponding dummy argument on invocation of the procedure, and
- when execution of the procedure completes, any pointers that do not become undefined (19.5.2.5) and are associated with the dummy argument remain associated with the actual argument.

6 If a dummy argument has INTENT (OUT) or INTENT (INOUT), the actual argument shall be definable. If a dummy argument has INTENT (OUT) and its associated actual argument is allocated, the actual argument is deallocated on procedure invocation (9.7.3.2).

15.5.2.7 Pointer dummy variables

1 The requirements in this subclause apply to actual arguments that correspond to dummy data pointers.

C1541 The actual argument corresponding to a dummy pointer with the CONTIGUOUS attribute shall be simply contiguous (9.5.4).

C1542 The actual argument corresponding to a dummy pointer shall not be a coindexed object.

2 If the dummy argument does not have INTENT (IN), the actual argument shall be a pointer. Otherwise, the actual argument shall be a pointer or a valid target for the dummy pointer in a pointer assignment statement. If the actual argument is not a pointer, the dummy pointer becomes pointer associated with the actual argument.

3 If the dummy argument has INTENT (OUT), the pointer association status of the actual argument becomes undefined on invocation of the procedure.
NOTE 15.27
For more explanatory information on pointers as dummy arguments, see subclause C.10.4.

15.5.2.8 Coarray dummy variables

1 If the dummy argument is a coarray, the corresponding actual argument shall be a coarray and shall have the VOLATILE attribute if and only if the dummy argument has the VOLATILE attribute.

2 If the dummy argument is an array coarray that has the CONTIGUOUS attribute or is not of assumed shape, the corresponding actual argument shall be simply contiguous or an element of a simply contiguous array.

NOTE 15.28
Consider the invocation of a procedure on a particular image. Each dummy coarray is associated with its ultimate argument on the image. In addition, during this execution of the procedure, this image can access the coarray corresponding to the ultimate argument on any other image. For example, consider

INTERFACE
  SUBROUTINE SUB(X)
    REAL :: X[*]
  END SUBROUTINE SUB
END INTERFACE
...
REAL :: A(1000)[*]
...
CALL SUB(A(10))

During execution of this invocation of SUB, the executing image has access through the syntax X[P] to A(10) on image P.

NOTE 15.29
Each invocation of a procedure with a nonallocatable coarray dummy argument establishes a dummy coarray for the image with its own bounds and cobounds. During this execution of the procedure, this image can use its own bounds and cobounds to access the coarray corresponding to the ultimate argument on any other image. For example, consider

INTERFACE
  SUBROUTINE SUB(X,N)
    INTEGER :: N
    REAL :: X(N,N)[N,*]
  END SUBROUTINE SUB
END INTERFACE
...
REAL :: A(1000)[*]
...
CALL SUB(A,10)

During execution of this invocation of SUB, the executing image has access through the syntax X(1,2)[3,4] to A(11) on the image with image index 33.

NOTE 15.30
The requirements on an actual argument that corresponds to a dummy coarray that is not of assumed-shape or has the CONTIGUOUS attribute are designed to avoid forcing a processor to use the so-called copy-in/copy-out argument passing mechanism.
15.5.2.9 Actual arguments associated with dummy procedure entities

1 If the interface of a dummy procedure is explicit, its characteristics as a procedure (15.3.1) shall be the same as those of its effective argument, except that a pure effective argument may be associated with a dummy argument that is not pure and an elemental intrinsic actual procedure may be associated with a dummy procedure (which cannot be elemental).

2 If the interface of a dummy procedure is implicit and either the dummy argument is explicitly typed or referenced as a function, it shall not be referenced as a subroutine and any corresponding actual argument shall be a function, function procedure pointer, or dummy procedure. If both the actual argument and dummy argument are known to be functions, they shall have the same type and type parameters. If only the dummy argument is known to be a function, the function that would be invoked by a reference to the dummy argument shall have the same type and type parameters, except that an external function with assumed character length may be associated with a dummy argument with explicit character length.

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

4 If a dummy argument is a dummy procedure without the POINTER attribute, its effective argument shall be an external, internal, module, or dummy procedure, or a specific intrinsic procedure listed in Table 16.2. If the specific name is also a generic name, only the specific procedure is associated with the dummy argument.

5 If a dummy argument is a procedure pointer, the corresponding actual argument shall be a procedure pointer, a reference to a function that returns a procedure pointer, a reference to the intrinsic function NULL, or a valid target for the dummy pointer in a pointer assignment statement. If the actual argument is not a pointer, the dummy argument shall have INTENT (IN) and becomes pointer associated with the actual argument.

6 When the actual argument is a procedure, the host instance of the dummy argument is the host instance of the actual argument (15.6.2.4).

7 If an external procedure or a dummy procedure is used as an actual argument, its interface shall be explicit or it shall be explicitly declared to have the EXTERNAL attribute.

15.5.2.10 Actual arguments and alternate return indicators

1 If a dummy argument is an asterisk (15.6.2.3), the corresponding actual argument shall be an alternate return specifier (R1525).

15.5.2.11 Sequence association

1 An actual argument represents an element sequence if it is an array expression, an array element designator, a default character scalar, or a scalar of type character with the C character kind (18.2.2). If the actual argument is an array expression, the element sequence consists of the elements in array element order. If the actual argument is an array element designator, the element sequence consists of that array element and each element that follows it in array element order.

2 If the actual argument is default character or of type character with the C character kind, and is an array expression, array element, or array element substring designator, the element sequence consists of the storage units beginning with the first storage unit of the actual argument and continuing to the end of the array. The storage units of an array element substring designator are viewed as array elements consisting of consecutive groups of storage units having the character length of the dummy array.

3 If the actual argument is default character or of type character with the C character kind, and is a scalar that is not an array element or array element substring designator, the element sequence consists of the storage units of the actual argument.

NOTE 15.31
Some of the elements in the element sequence might consist of storage units from different elements of the original array.
An actual argument that represents an element sequence and corresponds to a dummy argument that is an array is sequence associated with the dummy argument if the dummy argument is an explicit-shape or assumed-size array. The rank and shape of the actual argument need not agree with the rank and shape of the dummy argument, but the number of elements in the dummy argument shall not exceed the number of elements in the element sequence of the actual argument. If the dummy argument is assumed-size, the number of elements in the dummy argument is exactly the number of elements in the element sequence.

15.5.2.12 Argument presence and restrictions on arguments not present

A dummy argument or an entity that is host associated with a dummy argument is not present if the dummy argument

- does not correspond to an actual argument,
- corresponds to an actual argument that is not present, or
- does not have the ALLOCATABLE or POINTER attribute, and corresponds to an actual argument that
  - has the ALLOCATABLE attribute and is not allocated, or
  - has the POINTER attribute and is disassociated;

otherwise, it is present.

A nonoptional dummy argument shall be present. If an optional nonpointer dummy argument corresponds to a present pointer actual argument, the pointer association status of the actual argument shall not be undefined.

An optional dummy argument that is not present is subject to the following restrictions.

(1) If it is a data object, it shall not be referenced or be defined. If it is of a type that has default initialization, the initialization has no effect.
(2) It shall not be used as the data-target or proc-target of a pointer assignment.
(3) If it is a procedure or procedure pointer, it shall not be invoked.
(4) It shall not be supplied as an actual argument corresponding to a nonoptional dummy argument other than as the argument of the intrinsic function PRESENT or as an argument of a function reference that is a constant expression.
(5) A designator with it as the base object and with one or more subobject selectors shall not be supplied as an actual argument.
(6) If it is an array, it shall not be supplied as an actual argument to an elemental procedure unless an array of the same rank is supplied as an actual argument corresponding to a nonoptional dummy argument of that elemental procedure.
(7) If it is a pointer, it shall not be allocated, deallocated, nullified, pointer-assigned, or supplied as an actual argument corresponding to an optional nonpointer dummy argument.
(8) If it is allocatable, it shall not be allocated, deallocated, or supplied as an actual argument corresponding to an optional nonallocatable dummy argument.
(9) If it has length type parameters, they shall not be the subject of an inquiry.
(10) It shall not be used as the selector in an ASSOCIATE, SELECT RANK, or SELECT TYPE construct.
(11) It shall not be supplied as the data-ref in a procedure-designator.
(12) If it shall not be supplied as the scalar-variable in a proc-component-ref.

Except as noted in the list above, it may be supplied as an actual argument corresponding to an optional dummy argument, which is then also considered not to be present.

15.5.2.13 Restrictions on entities associated with dummy arguments

While an entity is associated with a dummy argument, the following restrictions hold.

(1) Action that affects the allocation status of the entity or a subobject thereof shall be taken through the dummy argument.
(2) If the allocation status of the entity or a subobject thereof is affected through the dummy argument, then at any time during the invocation and execution of the procedure, either before or after the allocation or deallocation, it shall be referenced only through the dummy argument.

(3) Action that affects the value of the entity or any subobject of it shall be taken only through the dummy argument unless
(a) the dummy argument has the POINTER attribute,
(b) the dummy argument is a scalar, assumed-shape, or assumed-rank object, and has the TARGET attribute but not the INTENT (IN) or CONTIGUOUS attributes, and the actual argument is a target other than a coindexed object or an array section with a vector subscript,
(c) the dummy argument is an assumed-rank object with the TARGET attribute and not the INTENT (IN) attribute, and the actual argument is a scalar target, or
(d) the dummy argument is a coarray and the action is a coindexed definition of the corresponding ultimate argument coarray by a different image.

(4) If the value of the entity or any subobject of it is affected through the dummy argument, then at any time during the invocation and execution of the procedure, either before or after the definition, it may be referenced only through that dummy argument unless
(a) the dummy argument has the POINTER attribute,
(b) the dummy argument is a scalar, assumed-shape, or assumed-rank object, and has the TARGET attribute but not the INTENT (IN) or CONTIGUOUS attributes, and the actual argument is a target other than a coindexed object or an array section with a vector subscript,
(c) the dummy argument is an assumed-rank object with the TARGET attribute and not the INTENT (IN) attribute, and the actual argument is a scalar target, or
(d) the dummy argument is a coarray and the reference is a coindexed reference of its corresponding ultimate argument coarray by a different image.

NOTE 15.32

In

SUBROUTINE OUTER
    REAL, POINTER :: A (:)
    ...
    ALLOCATE (A (1:N))
    ...
    CALL INNER (A)
    ...
END SUBROUTINE OUTER

CONTAINS
    SUBROUTINE INNER (B)
        REAL :: B (:)
        ...
    END SUBROUTINE INNER

    SUBROUTINE SET (C, D)
        REAL, INTENT (OUT) :: C
        REAL, INTENT (IN) :: D
        C = D
    END SUBROUTINE SET
END SUBROUTINE OUTER

an assignment statement such as

A (1) = 1.0

would not be permitted during the execution of INNER because this would be changing A without using B, but statements such as
NOTE 15.32  (cont.)

B (1) = 1.0

or

CALL SET (B (1), 1.0)

would be allowed. Similarly,

DEALLOCATE (A)

would not be allowed because this affects the allocation of B without using B. In this case,

DEALLOCATE (B)

also would not be permitted. If B were declared with the POINTER attribute, either of the statements

DEALLOCATE (A)

and

DEALLOCATE (B)

would be permitted, but not both.

NOTE 15.33

If there is a partial or complete overlap between the effective arguments of two different dummy arguments of the same procedure and the dummy arguments have neither the POINTER nor TARGET attribute, the overlapped portions cannot be defined, redefined, or become undefined during the execution of the procedure. For example, in

CALL SUB (A (1:5), A (3:9))

the array section A (3:5) cannot be defined, redefined, or become undefined through the first dummy argument because it is part of the argument associated with the second dummy argument and cannot be defined, redefined, or become undefined through the second dummy argument because it is part of the argument associated with the first dummy argument. The array section A (1:2) remains definable through the first dummy argument and A (6:9) remains definable through the second dummy argument.

This restriction applies equally to pointer targets. In

REAL, DIMENSION (10), TARGET :: A
REAL, DIMENSION (:), POINTER :: B, C
B => A (1:5)
C => A (3:9)
CALL SUB (B, C)  ! The dummy arguments of SUB are neither pointers nor targets.

the array section B (3:5) cannot be defined because it is part of the argument associated with the second dummy argument. The array section C (1:3) cannot be defined because it is part of the argument associated with the first dummy argument. The array section A (1:2), which is associated with B (1:2), remains definable through the first dummy argument and A (6:9), which is associated with C (4:7), remains definable through the second dummy argument.

NOTE 15.34

In
NOTE 15.34 (cont.)

```fortran
MODULE DATA
  REAL :: W, X, Y, Z
END MODULE DATA

PROGRAM MAIN
  USE DATA
  ... CALL INIT (X) ...
END PROGRAM MAIN
SUBROUTINE INIT (V)
  USE DATA
  ... READ (*, *) V ...
END SUBROUTINE INIT
```

variable X cannot be directly referenced at any time during the execution of INIT because it is being defined through the dummy argument V. X can be (indirectly) referenced through V. W, Y, and Z can be directly referenced. X can, of course, be directly referenced once execution of INIT is complete.

NOTE 15.35

The restrictions on entities associated with dummy arguments are intended to facilitate a variety of optimizations in the translation of the subprogram, including implementations of argument association in which the value of an actual argument that is neither a pointer nor a target is maintained in a register or in local storage.

NOTE 15.36

The exception to the aliasing restrictions for dummy coarrays enables cross-image access while the procedure is executing. Because nonatomic accesses from different images typically need to be separated by an image control statement, code optimization within segments is not unduly inhibited.

15.5.3 Function reference

A function is invoked during expression evaluation by a function-reference or by a defined operation (10.1.6). When it is invoked, all actual argument expressions are evaluated, then the arguments are associated, and then the function is executed. When execution of the function is complete, the value of the function result is available for use in the expression that caused the function to be invoked. The characteristics of the function result (15.3.3) are determined by the interface of the function. If a reference to an elemental function (15.8) is an elemental reference, all array arguments shall have the same shape.

15.5.4 Subroutine reference

A subroutine is invoked by execution of a CALL statement, execution of a defined assignment statement (10.2.1.4), defined input/output (12.6.4.8.3), or finalization (7.5.6). When a subroutine is invoked, all actual argument expressions are evaluated, then the arguments are associated, and then the subroutine is executed. When the actions specified by the subroutine are completed, the execution of the CALL statement, the execution of the defined assignment statement, the processing of an input or output list item, or finalization of an object is also completed. If a CALL statement includes one or more alternate return specifiers among its arguments, a branch to one of the statements indicated might occur, depending on the action specified by the subroutine. If a reference to an elemental subroutine (15.8) is an elemental reference, at least one actual argument shall correspond to an INTENT (OUT) or INTENT (INOUT) dummy argument, all such actual arguments shall be arrays, and all actual arguments shall be conformable.
15.5.5 Resolving named procedure references

15.5.5.1 Establishment of procedure names

1 The rules for interpreting a procedure reference depend on whether the procedure name in the reference is established by the available declarations and specifications to be generic in the scoping unit containing the reference, is established to be only specific in the scoping unit containing the reference, or is not established.

2 A procedure name is established to be generic in a scoping unit

(1) if that scoping unit contains an interface block with that name;
(2) if that scoping unit contains a GENERIC statement with a generic-spec that is that name;
(3) if that scoping unit contains an INTRINSIC attribute specification for that name and it is the generic name of an intrinsic procedure;
(4) if that scoping unit contains a USE statement that makes that procedure name accessible and the corresponding name in the module is established to be generic; or
(5) if that scoping unit contains no declarations of that name, that scoping unit has a host scoping unit, and that name is established to be generic in the host scoping unit.

3 A procedure name is established to be only specific in a scoping unit if it is established to be specific and not established to be generic. It is established to be specific

(1) if that scoping unit contains a module subprogram, internal subprogram, or statement function statement that defines a procedure with that name;
(2) if that scoping unit is of a subprogram that defines a procedure with that name;
(3) if that scoping unit contains an INTRINSIC attribute specification for that name and it is the name of a specific intrinsic procedure;
(4) if that scoping unit contains an explicit EXTERNAL attribute specification for that name;
(5) if that scoping unit contains a USE statement that makes that procedure name accessible and the corresponding name in the module is established to be specific; or
(6) if that scoping unit contains no declarations of that name, that scoping unit has a host scoping unit, and that name is established to be specific in the host scoping unit.

4 A procedure name is not established in a scoping unit if it is neither established to be generic nor established to be specific.

15.5.5.2 Resolving procedure references to names established to be generic

1 If the reference is consistent with a nonelemental reference to one of the specific interfaces of a generic interface that has that name and either is defined in the scoping unit in which the reference appears or is made accessible by a USE statement in the scoping unit, the reference is to the specific procedure in the interface block that provides that interface. The rules in 15.4.3.4.5 ensure that there can be at most one such specific procedure.

2 Otherwise, if the reference is consistent with an elemental reference to one of the specific interfaces of a generic interface that has that name and either is defined in the scoping unit in which the reference appears or is made accessible by a USE statement in the scoping unit, the reference is to the specific elemental procedure in the interface block that provides that interface. The rules in 15.4.3.4.5 ensure that there can be at most one such specific elemental procedure.

3 Otherwise, if the scoping unit contains either an INTRINSIC attribute specification for that name or a USE statement that makes that name accessible from a module in which the corresponding name is specified to have the INTRINSIC attribute, and if the reference is consistent with the interface of that intrinsic procedure, the reference is to that intrinsic procedure.

4 Otherwise, if the scoping unit has a host scoping unit, the name is established to be generic in that host scoping unit, and there is agreement between the scoping unit and the host scoping unit as to whether the name is a
function name or a subroutine name, the name is resolved by applying the rules in this subclause to the host scoping unit as if the reference appeared there.

Otherwise, if the name is that of an intrinsic procedure and the reference is consistent with that intrinsic procedure, the reference is to that intrinsic procedure.

NOTE 15.37

These rules allow particular specific procedures with the same generic identifier to be used for particular array ranks and a general elemental version to be used for other ranks. For example, given an interface block such as:

```
INTERFACE RANF
  ELEMENTAL FUNCTION SCALAR_RANF(X)
    REAL, INTENT(IN) :: X
  END FUNCTION SCALAR_RANF
  FUNCTION VECTOR_RANF(X)
    REAL X(:)
    REAL VECTOR_RANF(SIZE(X))
  END FUNCTION VECTOR_RANF
END INTERFACE RANF
```

and a declaration such as:

```
REAL A(10,10), AA(10,10)
```

then the statement

```
A = RANF(AA)
```

is an elemental reference to SCALAR_RANF. The statement

```
A(6:10,2) = RANF(AA(6:10,2))
```

is a nonelemental reference to VECTOR_RANF.

NOTE 15.38

In the USE statement case, it is possible, because of the renaming facility, for the name in the reference to be different from the name of the intrinsic procedure.

15.5.5.3 Resolving procedure references to names established to be only specific

1 If the name has the EXTERNAL attribute,
   • if it is a procedure pointer, the reference is to its target;
   • if it is a dummy procedure that is not a procedure pointer, the reference is to the effective argument corresponding to that name;
   • otherwise, the reference is to the external procedure with that name.

2 If the name is that of an accessible external procedure, internal procedure, module procedure, intrinsic procedure, or statement function, the reference is to that procedure.

NOTE 15.39

Because of the renaming facility of the USE statement, the name in the reference can be different from the original name of the procedure.
15.5.5.4 Resolving procedure references to names not established

1 If the name is the name of a dummy argument of the scoping unit, the dummy argument is a dummy procedure and the reference is to that dummy procedure. That is, the procedure invoked by executing that reference is the effective argument corresponding to that dummy procedure.

2 Otherwise, if the name is the name of an intrinsic procedure, and if there is agreement between the reference and the status of the intrinsic procedure as being a function or subroutine, the reference is to that intrinsic procedure.

3 Otherwise, the reference is to an external procedure with that name.

15.5.6 Resolving type-bound procedure references

1 If the binding-name in a procedure-designator (R1522) is that of a specific type-bound procedure, the procedure referenced is the one bound to that name in the dynamic type of the data-ref.

2 If the binding-name in a procedure-designator is that of a generic type bound procedure, the generic binding with that name in the declared type of the data-ref is used to select a specific binding using the following criteria.

- If the reference is consistent with one of the specific bindings of that generic binding, that specific binding is selected.
- Otherwise, the reference shall be consistent with an elemental reference to one of the specific bindings of that generic binding; that specific binding is selected.

3 The reference is to the procedure bound to the same name as the selected specific binding in the dynamic type of the data-ref.

15.6 Procedure definition

15.6.1 Intrinsic procedure definition

1 Intrinsic procedures are defined as an inherent part of the processor. A standard-conforming processor shall include the intrinsic procedures described in Clause 16, but may include others. However, a standard-conforming program shall not make use of intrinsic procedures other than those described in Clause 16.

15.6.2 Procedures defined by subprograms

15.6.2.1 General

1 A procedure is defined by the initial SUBROUTINE or FUNCTION statement of a subprogram, and each ENTRY statement defines an additional procedure (15.6.2.6).

2 A subprogram is specified to be elemental (15.8), pure (15.7), recursive, or a separate module subprogram (15.6.2.5) by a prefix in its initial SUBROUTINE or FUNCTION statement.

R1526 prefix is prefix-spec [ prefix-spec ] ...
R1527 prefix-spec is declaration-type-spec or ELEMENTAL or IMPURE or MODULE or NON_RECURSIVE or PURE or RECURSIVE

C1543 (R1526) A prefix shall contain at most one of each prefix-spec.
A prefix shall not specify both PURE and IMPURE.

A prefix shall not specify both NON_RECURSIVE and RECURSIVE.

An elemental procedure shall not have the BIND attribute.

MODULE shall appear only in the function-stmt or subroutine-stmt of a module subprogram or of a nonabstract interface body that is declared in the scoping unit of a module or submodule.

If MODULE appears in the prefix of a module subprogram, it shall have been declared to be a separate module procedure in the containing program unit or an ancestor of that program unit.

If MODULE appears in the prefix of a module subprogram, the subprogram shall specify the same characteristics and dummy argument names as its corresponding module procedure interface body.

If MODULE appears in the prefix of a module subprogram and a binding label is specified, it shall be the same as the binding label specified in the corresponding module procedure interface body.

If MODULE appears in the prefix of a module subprogram, NON_RECURSIVE shall appear if and only if NON_RECURSIVE appears in the prefix in the corresponding module procedure interface body.

The NON_RECURSIVE prefix-spec shall not appear if any procedure defined by the subprogram directly or indirectly invokes itself or any other procedure defined by the subprogram.

If the prefix-spec PURE appears, or the prefix-spec ELEMENTAL appears and IMPURE does not appear, the subprogram is a pure subprogram and shall meet the additional constraints of 15.7.

If the prefix-spec ELEMENTAL appears, the subprogram is an elemental subprogram and shall meet the additional constraints of 15.8.1.

A proc-language-binding-spec is language-binding-spec

A proc-language-binding-spec with a NAME= specifier shall not be specified in the function-stmt or subroutine-stmt of an internal procedure, or of an interface body for an abstract interface or a dummy procedure.

If proc-language-binding-spec is specified for a function, the function result shall be an interoperable scalar variable.

If proc-language-binding-spec is specified for a procedure, each of its dummy arguments shall be an interoperable procedure (18.3.7) or a variable that is interoperable (18.3.5, 18.3.6), assumed-shape, assumed-rank, assumed-type, of type CHARACTER with assumed length, or that has the ALLOCATABLE or POINTER attribute.

A variable that is a dummy argument of a procedure that has a proc-language-binding-spec shall be assumed-type or of interoperable type.

The ALLOCATABLE or POINTER attribute shall not be specified for a default-initialized dummy argument of a procedure that has a proc-language-binding-spec.

If proc-language-binding-spec is specified for a procedure, it shall not have a dummy argument that is a coarray.

If proc-language-binding-spec is specified for a procedure, it shall not have an array dummy argument with the VALUE attribute.

15.6.2.2 Function subprogram

A function subprogram is a subprogram that has a FUNCTION statement as its first statement.
R1529  function-subprogram is function-stmt
  [ specification-part ]
  [ execution-part ]
  [ internal-subprogram-part ]
end-function-stmt

R1530  function-stmt is [ prefix ] FUNCTION function-name ■
  ■ ( [ dummy-arg-name-list ] ) [ suffix ]

C1559 (R1530) If RESULT appears, result-name shall not be the same as function-name and shall not be the same as the entry-name in any ENTRY statement in the subprogram.

C1560 (R1530) If RESULT appears, the function-name shall not appear in any specification statement in the scoping unit of the function subprogram.

R1531  dummy-arg-name is name

C1561 (R1531) A dummy-arg-name shall be the name of a dummy argument.

R1532  suffix is proc-language-binding-spec [ RESULT ( result-name ) ]
  or RESULT ( result-name ) [ proc-language-binding-spec ]

R1533  end-function-stmt is END [ FUNCTION [ function-name ] ]

C1562 (R1529) An internal function subprogram shall not contain an internal-subprogram-part.

C1563 (R1533) If a function-name appears in the end-function-stmt, it shall be identical to the function-name specified in the function-stmt.

2  The name of the function is function-name.

3  The type and type parameters (if any) of the result of the function defined by a function subprogram may be specified by a type specification in the FUNCTION statement or by the name of the function result appearing in a type declaration statement in the specification part of the function subprogram. They shall not be specified both ways. If they are not specified either way, they are determined by the implicit typing rules in effect within the function subprogram. If the function result is an array, allocatable, or a pointer, this shall be specified by specifications of the name of the function result within the function body. The specifications of the function result attributes, the specification of dummy argument attributes, and the information in the procedure heading collectively define the characteristics of the function (15.3.1).

4  If RESULT appears, the name of the function result of the function is result-name and all occurrences of the function name in execution-part statements in its scope refer to the function itself. If RESULT does not appear, the name of the function result is function-name and all occurrences of the function name in execution-part statements in its scope are references to the function result. On completion of execution of the function, the value returned is that of its function result. If the function result is a data pointer, the shape of the value returned by the function is determined by the shape of the function result when the execution of the function is completed. If the function result is not a pointer, its value shall be defined by the function. If the function result is a pointer, on return the pointer association status of the function result shall not be undefined.

NOTE 15.40
The function result is similar to any other entity (variable or procedure pointer) local to a function subprogram. Its existence begins when execution of the function is initiated and ends when execution of the function is terminated. However, because the final value of this entity is used subsequently in the evaluation of the expression that invoked the function, an implementation might defer releasing the storage occupied by that entity until after its value has been used in expression evaluation.
NOTE 15.41

The following is an example of the declaration of an interface body with the BIND attribute, and a reference to the procedure declared.

USE, INTRINSIC :: ISO_C_BINDING

INTERFACE
  FUNCTION JOE (I, J, R) BIND(C, NAME="FrEd")
    USE, INTRINSIC :: ISO_C_BINDING
    INTEGER(C_INT) :: JOE
    INTEGER(C_INT), VALUE :: I, J
    REAL(C_FLOAT), VALUE :: R
  END FUNCTION JOE
END INTERFACE

INT = JOE(1_C_INT, 3_C_INT, 4.0_C_FLOAT)
END PROGRAM

The invocation of the function JOE results in a reference to a function with a binding label "FrEd". FrEd could be a C function described by the C prototype

```c
int FrEd(int n, int m, float x);
```

15.6.2.3 Subroutine subprogram

A subroutine subprogram is a subprogram that has a SUBROUTINE statement as its first statement.

R1534 subroutine-subprogram is subroutine-stmt
  [ specification-part ]
  [ execution-part ]
  [ internal-subprogram-part ]
end-subroutine-stmt

R1535 subroutine-stmt is [ prefix ] SUBROUTINE subroutine-name ■
  ■ [ ( [ dummy-arg-list ] ) [ proc-language-binding-spec ] ]

C1564 (R1535) The prefix of a subroutine-stmt shall not contain a declaration-type-spec.

R1536 dummy-arg is dummy-arg-name
  or *

R1537 end-subroutine-stmt is END [ SUBROUTINE [ subroutine-name ] ]

C1565 (R1534) An internal subroutine subprogram shall not contain an internal-subprogram-part.

C1566 (R1537) If a subroutine-name appears in the end-subroutine-stmt, it shall be identical to the subroutine-name specified in the subroutine-stmt.

The name of the subroutine is subroutine-name.

15.6.2.4 Instances of a subprogram

When a procedure defined by a subprogram is invoked, an instance of that subprogram is created. Each instance has an independent sequence of execution and an independent set of dummy arguments, unsaved local variables, and unsaved local procedure pointers. Saved local entities are shared by all instances of the subprogram.

When a statement function is invoked, an instance of that statement function is created.
3 When execution of an instance completes it ceases to exist.

4 The caller of an instance of a procedure is the instance of the main program, subprogram, or statement function that invoked it. The call sequence of an instance of a procedure is its caller, followed by the call sequence of its caller. The call sequence of the main program is empty. The host instance of an instance of a statement function or an internal procedure that is invoked by its name is the first element of the call sequence that is an instance of the host of the statement function or internal subprogram. The host instance of an internal procedure that is invoked via a dummy procedure or procedure pointer is the host instance of the associating entity from when the argument association or pointer association was established (19.5.5). The host instance of a module procedure is the module or submodule in which it is defined. A main program or external subprogram has no host instance.

15.6.2.5 Separate module procedures

1 A separate module procedure is a module procedure defined by a separate-module-subprogram, by a function-subprogram whose initial statement contains the keyword MODULE, or by a subroutine-subprogram whose initial statement contains the keyword MODULE.

R1538 separate-module-subprogram is mp-subprogram-stmt

R1539 mp-subprogram-stmt is MODULE PROCEDURE procedure-name

C1567 (R1538) The procedure-name shall have been declared to be a separate module procedure in the containing program unit or an ancestor of that program unit.

C1568 (R1540) If a procedure-name appears in the end-mp-subprogram-stmt, it shall be identical to the procedure-name in the mp-subprogram-stmt.

2 A separate module procedure shall not be defined more than once.

3 The interface of a procedure defined by a separate-module-subprogram is explicitly declared by the mp-subprogram-stmt to be the same as its module procedure interface body. It is recursive if and only if it is declared to be recursive by the interface body, and if it is a function its result name is determined by the FUNCTION statement in the interface body.

NOTE 15.42

A separate module procedure can be accessed by use association only if its interface body is declared in the specification part of a module and is public.

15.6.2.6 ENTRY statement

1 An ENTRY statement permits a procedure reference to begin with a particular executable statement within the function or subroutine subprogram in which the ENTRY statement appears.

R1541 entry-stmt is ENTRY entry-name [ ( [ dummy-arg-list ] ) ] [ suffix ]

C1569 (R1541) If RESULT appears, the entry-name shall not appear in any specification or type declaration statement in the scoping unit of the function program.

C1570 (R1541) An entry-stmt shall appear only in an external-subprogram or a module-subprogram that does not define a separate module procedure. An entry-stmt shall not appear within an executable-construct.

C1571 (R1541) RESULT shall appear only if the entry-stmt is in a function subprogram.

C1572 (R1541) A dummy-arg shall not be an alternate return indicator if the ENTRY statement is in a function subprogram.
If RESULT appears, result-name shall not be the same as the function-name in the FUNCTION statement and shall not be the same as the entry-name in any ENTRY statement in the subprogram.

Optionally, a subprogram may have one or more ENTRY statements.

If the ENTRY statement is in a function subprogram, an additional function is defined by that subprogram. The name of the function is entry-name and the name of its result is result-name or is entry-name if no result-name is provided. The dummy arguments of the function are those specified in the ENTRY statement. If the characteristics of the result of the function named in the ENTRY statement are the same as the characteristics of the result of the function named in the FUNCTION statement, their result names identify the same entity, although their names need not be the same. Otherwise, they are storage associated and shall all be nonpointer, nonallocatable scalar variables that are default integer, default real, double precision real, default complex, or default logical.

If the ENTRY statement is in a subroutine subprogram, an additional subroutine is defined by that subprogram. The name of the subroutine is entry-name. The dummy arguments of the subroutine are those specified in the ENTRY statement.

The order, number, types, kind type parameters, and names of the dummy arguments in an ENTRY statement may differ from the order, number, types, kind type parameters, and names of the dummy arguments in the FUNCTION or SUBROUTINE statement in the containing subprogram.

Because an ENTRY statement defines an additional function or an additional subroutine, it is referenced in the same manner as any other function or subroutine (15.5).

In a subprogram, a dummy argument specified in an ENTRY statement shall not appear in an executable statement preceding that ENTRY statement, unless it also appears in a FUNCTION, SUBROUTINE, or ENTRY statement that precedes the executable statement. A function result specified by a result-name in an ENTRY statement shall not appear in any executable statement that precedes the first RESULT clause with that name.

In a subprogram, a dummy argument specified in an ENTRY statement shall not appear in the expression of a statement function that precedes the first dummy-arg with that name in the subprogram. A function result specified by a result-name in an ENTRY statement shall not appear in the expression of a statement function that precedes the first RESULT clause with that name.

If a dummy argument appears in an executable statement, the execution of the executable statement is permitted during the execution of a reference to the function or subroutine only if the dummy argument appears in the dummy argument list of the referenced procedure.

If a dummy argument is used in a specification expression to specify an array bound or character length of an object, the appearance of the object in a statement that is executed during a procedure reference is permitted only if the dummy argument appears in the dummy argument list of the referenced procedure and it is present (15.5.2.12).

The NON_RECURSIVE and RECURSIVE keywords are not used in an ENTRY statement. Instead, the presence or absence of NON_RECURSIVE in the initial SUBROUTINE or FUNCTION statement controls whether the procedure defined by an ENTRY statement is permitted to reference itself or another procedure defined by the subprogram.

The keywords PURE and IMPURE are not used in an ENTRY statement. Instead, the procedure defined by an ENTRY statement is pure if and only if the subprogram is a pure subprogram.

The keyword ELEMENTAL is not used in an ENTRY statement. Instead, the procedure defined by an ENTRY statement is elemental if and only if ELEMENTAL is specified in the SUBROUTINE or FUNCTION statement.

RETURN statement

RETURN [ scalar-int-expr ]

The return-stmt shall be in the inclusive scope of a function or subroutine subprogram.

The scalar-int-expr is allowed only in the inclusive scope of a subroutine subprogram.

Execution of the RETURN statement completes execution of the instance of the subprogram in which it appears. If the expression appears and has a value n between 1 and the number of asterisks in the dummy argument list, the CALL statement
that invoked the subroutine branches (11.2) to the branch target statement identified by the \( n \)th alternate return specifier in the actual argument list of the referenced procedure. If the expression is omitted or has a value outside the required range, there is no transfer of control to an alternate return.

2 Execution of an \texttt{end-function-stmt}, \texttt{end-mp-subprogram-stmt}, or \texttt{end-subroutine-stmt} is equivalent to execution of a RETURN statement with no expression.

15.6.2.8 CONTAINS statement

R1543 \texttt{contains-stmt} is CONTAINS

1 The CONTAINS statement separates the body of a main program, module, submodule, or subprogram from any internal or module subprograms it might contain, or it introduces the type-bound procedure part of a derived-type definition (7.5.5). The CONTAINS statement is not executable.

15.6.3 Definition and invocation of procedures by means other than Fortran

1 A procedure may be defined by means other than Fortran. The interface of a procedure defined by means other than Fortran may be specified by an interface body or procedure declaration statement. A reference to such a procedure is made as though it were defined by an external subprogram.

2 A procedure defined by means other than Fortran that is invoked by a Fortran procedure and does not cause termination of execution shall return to its caller.

NOTE 15.43
Examples of code that might cause a transfer of control that bypasses the normal return mechanism of a Fortran procedure are \texttt{setjmp} and \texttt{longjmp} in C and exception handling in other languages. No such behavior is permitted by this document.

3 If the interface of a procedure has a \texttt{proc-language-binding-spec}, the procedure is interoperable (18.10).

4 Interoperation with C functions is described in 18.10.

NOTE 15.44
For explanatory information on definition of procedures by means other than Fortran, see subclause C.10.2.

15.6.4 Statement function

1 A statement function is a function defined by a single statement.

R1544 \texttt{stmt-function-stmt} is \texttt{function-name ( [ dummy-arg-name-list ] ) = scalar-expr}

C1576 (R1544) Each \texttt{primary} in \texttt{scalar-expr} shall be a constant (literal or named), a reference to a variable, a reference to a function, or an expression in parentheses. Each operation shall be intrinsic. If \texttt{scalar-expr} contains a reference to a function, the reference shall not require an explicit interface, the function shall not require an explicit interface unless it is an intrinsic function, the function shall not be a transformational intrinsic, and the result shall be scalar. If an argument to a function is an array, it shall be an array name. If a reference to a statement function appears in \texttt{scalar-expr}, its definition shall have been provided earlier in the \texttt{scoping unit} and shall not be the name of the statement function being defined.

C1577 (R1544) Named constants in \texttt{scalar-expr} shall have been declared earlier in the \texttt{scoping unit} or made accessible by use or host association. If array elements appear in \texttt{scalar-expr}, the array shall have been declared as an array earlier in the \texttt{scoping unit} or made accessible by use or host association.

C1578 (R1544) If a \texttt{dummy-arg-name}, variable, function reference, or dummy function reference is typed by the implicit typing rules, its appearance in any subsequent \texttt{type declaration statement} shall confirm this implied type and the values of any implied type parameters.

C1579 (R1544) The \texttt{function-name} and each \texttt{dummy-arg-name} shall be specified, explicitly or implicitly, to be scalar.

C1580 (R1544) A given \texttt{dummy-arg-name} shall not appear more than once in any \texttt{dummy-arg-name-list}.
A statement function shall not be of a parameterized derived type.

The definition of a statement function with the same name as an accessible entity from the host shall be preceded by the declaration of its type in a type declaration statement.

The dummy arguments have a scope of the statement function statement. Each dummy argument has the same type and type parameters as the entity of the same name in the scoping unit containing the statement function statement.

A statement function shall not be supplied as an actual argument.

Execution of a statement function consists of evaluating the expression using the values of the actual arguments for the values of the corresponding dummy arguments and, if necessary, converting the result to the declared type and type parameters of the function.

A function reference in the scalar expression shall not cause a dummy argument of the statement function to become redefined or undefined.

15.7 Pure procedures

A pure procedure is

- a pure intrinsic procedure (16.1),
- a module procedure in an intrinsic module, if it is specified to be pure,
- defined by a pure subprogram,
- a dummy procedure that has been specified to be PURE, or
- a statement function that references only pure functions and does not contain the designator of a variable with the VOLATILE attribute.

A pure subprogram is a subprogram that has the prefix-spec PURE or that has the prefix-spec ELEMENTAL and does not have the prefix-spec IMPURE. The following additional constraints apply to pure subprograms.

The specification-part of a pure function subprogram shall specify that all its nonpointer dummy data objects have the INTENT (IN) or the VALUE attribute.

The function result of a pure function shall not be such that finalization of a reference to the function would reference an impure procedure.

A pure function shall not have a polymorphic allocatable result.

The specification-part of a pure subroutine subprogram shall specify the intents of all its nonpointer dummy data objects that do not have the VALUE attribute.

An INTENT (OUT) dummy argument of a pure procedure shall not be such that finalization of the actual argument would reference an impure procedure.

An INTENT (OUT) dummy argument of a pure procedure shall not be polymorphic or have a polymorphic allocatable ultimate component.

A local variable of a pure subprogram, or of a BLOCK construct within a pure subprogram, shall not have the SAVE or VOLATILE attribute.

NOTE 15.45

Variable initialization in a type-declaration-stmt or a data-stmt implies the SAVE attribute; therefore, such initialization is also disallowed.

The specification-part of a pure subprogram shall specify that all its dummy procedures are pure.

If a procedure that is neither an intrinsic procedure nor a statement function is used in a context that requires it to be pure, then its interface shall be explicit in the scope of that use. The interface shall specify that
the procedure is pure.

C1591 All internal subprograms in a pure subprogram shall be pure.

C1592 A designator of a variable with the VOLATILE attribute shall not appear in a pure subprogram.

C1593 In a pure subprogram any designator with a base object that is in common or accessed by host or use association, is a pointer dummy argument of a pure function, is a dummy argument with the INTENT (IN) attribute, is a coindexed object, or an object that is storage associated with any such variable, shall not be used

(1) in a variable definition context (19.6.7),
(2) in a pointer association context (19.6.8),
(3) as the data-target in a pointer-assignment-stmt,
(4) as the expr corresponding to a component in a structure-constructor if the component has the POINTER attribute or has a pointer component at any level of component selection,
(5) as the expr of an intrinsic assignment statement in which the variable is of a derived type if the derived type has a pointer component at any level of component selection,
(6) as the source-expr in a SOURCE= specifier if the designator is of a derived type that has a pointer component at any level of component selection,
(7) as an actual argument corresponding to a dummy argument with the POINTER attribute, or
(8) as the actual argument to the function C_LOC from the intrinsic module ISO_C_BINDING.

NOTE 15.46
Item 5 requires that processors be able to determine if entities with the PRIVATE attribute or with private components have a pointer component.

C1594 Any procedure referenced in a pure subprogram, including one referenced via a defined operation, defined assignment, defined input/output, or finalization, shall be pure.

C1595 A statement that might result in the deallocation of a polymorphic entity is not permitted in a pure procedure.

NOTE 15.47
This includes intrinsic assignment to a variable that has a potential subobject component that is polymorphic and allocatable.

C1596 A pure subprogram shall not contain a print-stmt, open-stmt, close-stmt, backspace-stmt, endfile-stmt, rewind-stmt, flush-stmt, wait-stmt, or inquire-stmt.

C1597 A pure subprogram shall not contain a read-stmt or write-stmt whose io-unit is a file-unit-number or *.

C1598 A pure subprogram shall not contain an image control statement (11.6.1).

NOTE 15.48
The above constraints are designed to guarantee that a pure procedure is free from side effects (modifications of data visible outside the procedure), which means that it is safe to reference it in constructs such as DO CONCURRENT and FORALL, where there is no explicit order of evaluation.

The constraints on pure subprograms appear to be complicated, but it is not necessary for a programmer to be intimately familiar with them. From the programmer’s point of view, these constraints can be summarized as follows: a pure subprogram shall not contain any operation that could conceivably result in an assignment or pointer assignment to a common variable, a variable accessed by use or host association, or an INTENT (IN) dummy argument; nor shall a pure subprogram contain any operation that could conceivably perform any external file input/output or STOP operation. Note the use of the word conceivably; it is not sufficient for a pure subprogram merely to be side-effect free in practice. For example, a function that
NOTE 15.48 (cont.)

contains an assignment to a global variable but in a block that is not executed in any invocation of the function is nevertheless not a pure function. The exclusion of functions of this nature is required if strict compile-time checking is to be used.

It is expected that most library procedures will conform to the constraints required of pure procedures, and so can be declared pure and referenced in DO CONCURRENT constructs, FORALL statements and constructs, and within user-defined pure procedures.

NOTE 15.49

Pure subroutines are included to allow subroutine calls from pure procedures in a safe way, and to allow forall-assignment-stmts to be defined assignments. The constraints for pure subroutines are based on the same principles as for pure functions, except that side effects to INTENT (OUT), INTENT (INOUT), and pointer dummy arguments are permitted.

15.8 Elemental procedures

15.8.1 Elemental procedure declaration and interface

An elemental procedure is an elemental intrinsic procedure or a procedure that is defined by an elemental subprogram. An elemental procedure has only scalar dummy arguments, but may have array actual arguments.

An elemental subprogram has the prefix-spec ELEMENTAL. An elemental subprogram is a pure subprogram unless it has the prefix-spec IMPURE. The following additional constraints apply to elemental subprograms.

C1599 All dummy arguments of an elemental procedure shall be scalar noncoarray dummy data objects and shall not have the POINTER or ALLOCATABLE attribute.

C15100 The result of an elemental function shall be scalar, and shall not have the POINTER or ALLOCATABLE attribute.

C15101 The specification-part of an elemental subprogram shall specify the intents of all of its dummy arguments that do not have the VALUE attribute.

C15102 In the specification-expr that specifies a type parameter value of the result of an elemental function, an object designator with a dummy argument of the function as the base object shall appear only as the subject of a specification inquiry (10.1.11), and that specification inquiry shall not depend on a property that is deferred.

In a reference to an elemental procedure, if any argument is an array, each actual argument that corresponds to an INTENT (OUT) or INTENT (INOUT) dummy argument shall be an array. All actual arguments shall be conformable. An array actual argument is considered to be associated with the scalar dummy arguments of the procedure throughout the entire execution of the elemental reference; thus, the restrictions on actions specified in 15.5.2.13 apply to the entirety of the actual array argument.

15.8.2 Elemental function actual arguments and results

If a generic name or a specific name is used to reference an elemental function, the shape of the result is the same as the shape of the actual argument with the greatest rank. If there are no actual arguments or the actual arguments are all scalar, the result is scalar. In the array case, the values of the elements, if any, of the result are the same as would have been obtained if the scalar function had been applied separately, in array element order, to corresponding elements of each array actual argument.
NOTE 15.50
An example of an elemental reference to the intrinsic function MAX:

if X and Y are arrays of shape (M, N),

\( \text{MAX}(X, 0.0, Y) \)

is an array expression of shape (M, N) whose elements have values

\( \text{MAX}(X(I, J), 0.0, Y(I, J)), I = 1, 2, \ldots, M, J = 1, 2, \ldots, N \)

15.8.3 Elemental subroutine actual arguments

In a reference to an elemental subroutine, if the actual arguments corresponding to INTENT (OUT) and INTENT (INOUT) dummy arguments are arrays, the values of the elements, if any, of the results are the same as would be obtained if the subroutine had been applied separately, in array element order, to corresponding elements of each array actual argument.
16 Intrinsic procedures and modules

16.1 Classes of intrinsic procedures

Intrinsic procedures are divided into eight classes: inquiry functions, elemental functions, transformational functions, elemental subroutines, pure subroutines, atomic subroutines, collective subroutines, and (impure) subroutines.

An intrinsic inquiry function is one whose result depends on the properties of one or more of its arguments instead of their values; in fact, these argument values may be undefined. Unless the description of an intrinsic inquiry function states otherwise, these arguments are permitted to be unallocated allocatable variables or pointers that are undefined or disassociated. An elemental intrinsic function is one that is specified for scalar arguments, but may be applied to array arguments as described in 15.8. All other intrinsic functions are transformational functions; they almost all have one or more array arguments or an array result. All standard intrinsic functions are pure.

An atomic subroutine is an intrinsic subroutine that performs an atomic action. The semantics of atomic actions are described in 16.5.

A collective subroutine is an intrinsic subroutine that performs a cooperative calculation on a team of images without requiring synchronization. The semantics of collective subroutines are described in 16.6.

The subroutine MOVE_ALLOC and the elemental subroutine MVBITS are pure. No other standard intrinsic subroutine is pure.

Generic names of standard intrinsic procedures are listed in 16.7. In most cases, generic functions accept arguments of more than one type and the type of the result is the same as the type of the arguments. Specific names of standard intrinsic functions with corresponding generic names are listed in 16.8.

If an intrinsic procedure is used as an actual argument to a procedure, its specific name shall be used and it may be referenced in the called procedure only with scalar arguments. If an intrinsic procedure does not have a specific name, it shall not be used as an actual argument (15.5.2.9).

Elemental intrinsic procedures behave as described in 15.8.

16.2 Arguments to intrinsic procedures

16.2.1 General rules

All intrinsic procedures may be invoked with either positional arguments or argument keywords (15.5). The descriptions in 16.7 through 16.9 give the argument keyword names and positional sequence for standard intrinsic procedures.

Many of the intrinsic procedures have optional arguments. These arguments are identified by the notation “optional” in the argument descriptions. In addition, the names of the optional arguments are enclosed in square brackets in description headings and in lists of procedures. The valid forms of reference for procedures with optional arguments are described in 15.5.2.

NOTE 16.1

The text CMPLX (X [, Y, KIND]) indicates that Y and KIND are both optional arguments. Valid reference forms include CMPLX(x), CMPLX(x, y), CMPLX(x, KIND=kind), CMPLX(x, y, kind), and CMPLX(KIND=kind, X=x, Y=y).
NOTE 16.2

Some intrinsic procedures impose additional requirements on their optional arguments. For example, SELECTED_REAL_KIND requires that at least one of its optional arguments be present, and RANDOM_SEED requires that at most one of its optional arguments be present.

The dummy arguments of the specific intrinsic procedures in 16.8 have INTENT (IN). The dummy arguments of the intrinsic procedures in 16.9 have INTENT (IN) if the intent is not stated explicitly.

The actual argument corresponding to an intrinsic function dummy argument named KIND shall be a scalar integer constant expression and its value shall specify a representation method for the function result that exists on the processor.

Intrinsic subroutines that assign values to arguments of type character do so in accordance with the rules of intrinsic assignment (10.2.1.3).

In a reference to the intrinsic subroutine MVBITS, the actual arguments corresponding to the TO and FROM dummy arguments may be the same variable and may be associated scalar variables or associated array variables all of whose corresponding elements are associated. Apart from this, the actual arguments in a reference to an intrinsic subroutine shall be such that the execution of the intrinsic subroutine would satisfy the restrictions of 15.5.2.13.

An argument to an intrinsic procedure other than ASSOCIATED, NULL, or PRESENT shall be a data object.

16.2.2 The shape of array arguments

Unless otherwise specified, the intrinsic inquiry functions accept array arguments for which the shape need not be defined. The shape of array arguments to transformational and elemental intrinsic functions shall be defined.

16.2.3 Mask arguments

Some array intrinsic functions have an optional MASK argument of type logical that is used by the function to select the elements of one or more arguments to be operated on by the function. Any element not selected by the mask need not be defined at the time the function is invoked.

The MASK affects only the value of the function, and does not affect the evaluation, prior to invoking the function, of arguments that are array expressions.

16.2.4 DIM arguments and reduction functions

Some array intrinsic functions are “reduction” functions; that is, they reduce the rank of an array by collapsing one dimension (or all dimensions, usually producing a scalar result). These functions have a DIM argument that can specify the dimension to be reduced.

The process of reducing a dimension usually combines the selected elements with a simple operation such as addition or an intrinsic function such as MAX, but more sophisticated reductions are also provided, e.g. by COUNT and MAXLOC.

16.3 Bit model

16.3.1 General

The bit manipulation procedures are described in terms of a model for the representation and behavior of bits on a processor.

For the purposes of these procedures, a bit is defined to be a binary digit $w$ located at position $k$ of a nonnegative
integer scalar object based on a model nonnegative integer defined by

\[ j = \sum_{k=0}^{z-1} w_k \times 2^k \]

and for which \( w_k \) may have the value 0 or 1. This defines a sequence of bits \( w_{z-1} \ldots w_0 \), with \( w_{z-1} \) the leftmost bit and \( w_0 \) the rightmost bit. The positions of bits in the sequence are numbered from right to left, with the position of the rightmost bit being zero. The length of a sequence of bits is \( z \). An example of a model number compatible with the examples used in 16.4 would have \( z = 32 \), thereby defining a 32-bit integer.

The interpretation of a negative integer as a sequence of bits is processor dependent.

The inquiry function BIT_SIZE provides the value of the parameter \( z \) of the model.

Effectively, this model defines an integer object to consist of \( z \) bits in sequence numbered from right to left from 0 to \( z - 1 \). This model is valid only in the context of the use of such an object as the argument or result of an intrinsic procedure that interprets that object as a sequence of bits. In all other contexts, the model defined for an integer in 16.4 applies. In particular, whereas the models are identical for \( r = 2 \) and \( w_{z-1} = 0 \), they do not correspond for \( r \neq 2 \) or \( w_{z-1} = 1 \) and the interpretation of bits in such objects is processor dependent.

### 16.3.2 Bit sequence comparisons

When bit sequences of unequal length are compared, the shorter sequence is considered to be extended to the length of the longer sequence by padding with zero bits on the left.

Bit sequences are compared from left to right, one bit at a time, until unequal bits are found or all bits have been compared and found to be equal. If unequal bits are found, the sequence with zero in the unequal position is considered to be less than the sequence with one in the unequal position. Otherwise the sequences are considered to be equal.

### 16.3.3 Bit sequences as arguments to INT and REAL

When a \texttt{boz-literal-constant} is the argument \( A \) of the intrinsic function INT or REAL,

- if the length of the sequence of bits specified by \( A \) is less than the size in bits of a scalar variable of the same type and kind type parameter as the result, the \texttt{boz-literal-constant} is treated as if it were extended to a length equal to the size in bits of the result by padding on the left with zero bits, and
- if the length of the sequence of bits specified by \( A \) is greater than the size in bits of a scalar variable of the same type and kind type parameter as the result, the \texttt{boz-literal-constant} is treated as if it were truncated from the left to a length equal to the size in bits of the result.

C1601 If a \texttt{boz-literal-constant} is truncated as an argument to the intrinsic function REAL, the discarded bits shall all be zero.

**NOTE 16.3**
The result values of the intrinsic functions CMPLX and DBLE are defined by references to the intrinsic function REAL with the same arguments. Therefore, the padding and truncation of \texttt{boz-literal-constant} arguments to those intrinsic functions is the same as for the intrinsic function REAL.

### 16.4 Numeric models

The numeric manipulation and inquiry functions are described in terms of a model for the representation and behavior of numbers on a processor. The model has parameters that are determined so as to make the model best fit the machine on which the program is executed.
The model set for integer $i$ is defined by

$$i = s \times \sum_{k=0}^{q-1} w_k \times r^k$$

where $r$ is an integer exceeding one, $q$ is a positive integer, each $w_k$ is a nonnegative integer less than $r$, and $s$ is $+1$ or $-1$.

The model set for real $x$ is defined by

$$x = \begin{cases} 
0 & \text{or} \\
 s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k} , 
\end{cases}$$

where $b$ and $p$ are integers exceeding one; each $f_k$ is a nonnegative integer less than $b$, with $f_1$ nonzero; $s$ is $+1$ or $-1$; and $e$ is an integer that lies between some integer maximum $e_{\text{max}}$ and some integer minimum $e_{\text{min}}$ inclusively. For $x = 0$, its exponent $e$ and digits $f_k$ are defined to be zero. The integer parameters $r$ and $q$ determine the set of model integers and the integer parameters $b$, $p$, $e_{\text{min}}$, and $e_{\text{max}}$ determine the set of model floating-point numbers.

The parameters of the integer and real models are available for each representation method of the integer and real types. The parameters characterize the set of available numbers in the definition of the model. Intrinsic functions provide the values of some parameters and other values related to the models.

There is also an extended model set for each kind of real $x$; this extended model is the same as the ordinary model except that there are no limits on the range of the exponent $e$.

### NOTE 16.4

Some of the function descriptions use the models

$$i = s \times \sum_{k=0}^{30} w_k \times 2^k$$

and

$$x = 0 \text{ or } s \times 2^e \times \left( \frac{1}{2} + \sum_{k=2}^{24} f_k \times 2^{-k} \right) , \quad -126 \leq e \leq 127$$

### 16.5 Atomic subroutines

An atomic subroutine is an intrinsic subroutine that performs an action on its ATOM argument, and if it has an OLD argument, determination of the value to be assigned to that argument, atomically. Definition or evaluation of any argument other than ATOM is not performed atomically.

For any two executions in unordered segments of atomic subroutines whose ATOM argument is the same object, the effect is as if one of the executions is performed completely before the other execution begins. Which execution is performed first is processor dependent. The sequence of atomic actions within ordered segments is specified in 5.3.5. If successive atomic subroutine invocations on image $P$ redefine a variable atomically in segments $P_i$ and $P_j$, atomic references to that variable from image $Q$ in a segment $Q_k$ that is unordered relative to $P_i$ and $P_j$ may observe the changes in the value of that variable in any order.

Atomic operations shall make asynchronous progress. If a variable $X$ on image $P$ is defined by an atomic subroutine on image $Q$, image $R$ repeatedly references $X$ [$P$] by an atomic subroutine in an unordered segment,
and no other image defines X \([P]\) in an unordered segment, image \(R\) shall eventually receive the value assigned
by image \(Q\), even if none of the images \(P, Q,\) or \(R\) execute an image control statement until after the definition
of X \([P]\) by image \(Q\) and the reception of that value by image \(R\).

4 If the STAT argument is present in an invocation of an atomic subroutine and no error condition occurs, it is
assigned the value zero.

5 If the STAT argument is present in an invocation of an atomic subroutine and an error condition occurs, it
is assigned a processor-dependent positive value, and any other argument that is not INTENT (IN) becomes
undefined.

6 If the STAT argument is not present in an invocation of an atomic subroutine and an error condition occurs,
error termination is initiated.

**NOTE 16.5**
The properties of atomic subroutines are intended to support custom synchronization mechanisms. The
program will need to handle all possible orderings of sequences of atomic subroutine executions that can
arise as a consequence of the above rules; note that the orderings can appear to be different on different
images even in the same program execution.

### 16.6 Collective subroutines

1 Successful execution of a collective subroutine performs a calculation on all the images of the current team and
assigns a computed value on one or all of them. If it is invoked by one image, it shall be invoked by the same
statement on all active images of its current team in segments that are not ordered with respect to each other;
corresponding references participate in the same collective computation.

2 Before execution of the first CHANGE TEAM statement on an image, in between executions of CHANGE
TEAM and/or END TEAM statements, and after the last execution of an END TEAM statement, the sequence
of invocations of collective subroutines shall be the same on all active images of a team. A collective subroutine
shall not be referenced when an image control statement is not permitted to be executed (for example, in a
procedure invoked from a CRITICAL construct).

C1602 A reference to a collective subroutine shall not appear in a context where an image control statement is
not permitted to appear.

3 If the A argument in a reference to a collective subroutine is a coarray, the corresponding ultimate arguments on
all active images of the current team shall be corresponding coarrays as described in 5.4.7.

4 If the optional STAT argument is present in a reference to a collective subroutine on one image, it shall be present
on all the corresponding references.

5 If the STAT argument is present in an invocation of a collective subroutine on one image, its execution is successful, it is
assigned the value zero.

6 If the STAT argument is present in an invocation of a collective subroutine and an error condition occurs, it is
assigned a nonzero value and the A argument becomes undefined.

7 If the STAT argument is present and the current team contains stopped images, an error condition occurs and
the value STAT_STOPPED_IMAGE from the intrinsic module ISO_FORTRAN_ENV is assigned to the STAT
argument. Otherwise, if the current team contained failed images, an error condition occurs and the value
STAT_FAILED_IMAGE from the intrinsic module ISO_FORTRAN_ENV is assigned to the STAT argument.
If any other error condition occurs, the STAT argument is assigned a processor-dependent positive value that is
different from the value of STAT_STOPPED_IMAGE or STAT_FAILED_IMAGE.

8 If the STAT argument is not present in an invocation of a collective subroutine and an error condition occurs,
error termination is initiated.
If the ERRMSG argument is present in an invocation of a **collective subroutine** and an error condition occurs, it is assigned an explanatory message, truncated or padded with blanks if required. If no error condition occurs, the definition status and value of ERRMSG are unchanged.

**NOTE 16.6**
The argument A becomes undefined if an error condition occurs during execution of a **collective subroutine** because it is intended to allow the processor to use A for intermediate values during calculation.

**NOTE 16.7**
Although the calculations performed by a collective subroutine have some internal synchronizations, a reference to a collective subroutine is not an **image control statement**.

### 16.7 Standard generic intrinsic procedures

For all of the standard intrinsic procedures, the arguments shown are the names that shall be used for **argument keywords** if the keyword form is used for **actual arguments**.

**NOTE 16.8**
For example, a reference to CMPLX can be written in the form CMPLX (A, B, M) or in the form CMPLX (Y = B, KIND = M, X = A).

**NOTE 16.9**
Many of the **argument keywords** have names that are indicative of their usage. For example:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KIND</td>
<td>Describes the kind type parameter of the result</td>
</tr>
<tr>
<td>STRING, STRING_A</td>
<td>An arbitrary character string</td>
</tr>
<tr>
<td>BACK</td>
<td>Controls the direction of string scan (forward or backward)</td>
</tr>
<tr>
<td>MASK</td>
<td>A mask to be applied to the arguments</td>
</tr>
<tr>
<td>DIM</td>
<td>A selected dimension of an array argument</td>
</tr>
</tbody>
</table>

In the Class column of Table 16.1,

- A indicates that the procedure is an **atomic subroutine**,
- C indicates that the procedure is a **collective subroutine**,
- E indicates that the procedure is an **elemental function**,
- ES indicates that the procedure is an **elemental subroutine**,
- I indicates that the procedure is an **inquiry function**,
- PS indicates that the procedure is a **pure subroutine**,
- S indicates that the procedure is an impure subroutine, and
- T indicates that the procedure in a **transformational function**.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Arguments</th>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>(A)</td>
<td>E</td>
<td>Absolute value.</td>
</tr>
<tr>
<td>ACHAR</td>
<td>(I [, KIND])</td>
<td>E</td>
<td>Character from ASCII code value.</td>
</tr>
<tr>
<td>ACOS</td>
<td>(X)</td>
<td>E</td>
<td>Arccosine (inverse cosine) function.</td>
</tr>
<tr>
<td>ACOSH</td>
<td>(X)</td>
<td>E</td>
<td>Inverse hyperbolic cosine function.</td>
</tr>
<tr>
<td>ADJUSTL</td>
<td>(STRING)</td>
<td>E</td>
<td>Left-justified string value.</td>
</tr>
<tr>
<td>ADJUSTR</td>
<td>(STRING)</td>
<td>E</td>
<td>Right-justified string value.</td>
</tr>
<tr>
<td>AIMAG</td>
<td>(Z)</td>
<td>E</td>
<td>Imaginary part of a complex number.</td>
</tr>
<tr>
<td>AINT</td>
<td>(A [, KIND])</td>
<td>E</td>
<td>Truncation toward 0 to a whole number.</td>
</tr>
</tbody>
</table>
Table 16.1: Standard generic intrinsic procedure summary  

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Arguments</th>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>(MASK) or (MASK, DIM)</td>
<td>T</td>
<td>Array reduced by .AND. operator.</td>
</tr>
<tr>
<td>ALLOCATED</td>
<td>(ARRAY) or (SCALAR)</td>
<td>I</td>
<td>Allocation status of allocatable variable.</td>
</tr>
<tr>
<td>ANINT</td>
<td>(A [, KIND])</td>
<td>E</td>
<td>Nearest whole number.</td>
</tr>
<tr>
<td>ANY</td>
<td>(MASK) or (MASK, DIM)</td>
<td>T</td>
<td>Array reduced by .OR. operator.</td>
</tr>
<tr>
<td>ASIN</td>
<td>(X)</td>
<td>E</td>
<td>Arcsine (inverse sine) function.</td>
</tr>
<tr>
<td>ASINH</td>
<td>(X)</td>
<td>E</td>
<td>Inverse hyperbolic sine function.</td>
</tr>
<tr>
<td>ASSOCIATED</td>
<td>(POINTER [, TARGET])</td>
<td>I</td>
<td>Pointer (inverse sine) function.</td>
</tr>
<tr>
<td>ATAN</td>
<td>(X) or (Y, X)</td>
<td>E</td>
<td>Arctangent (inverse tangent) function.</td>
</tr>
<tr>
<td>ATAN2</td>
<td>(Y, X)</td>
<td>E</td>
<td>Arctangent (inverse tangent) function.</td>
</tr>
<tr>
<td>ATANH</td>
<td>(X)</td>
<td>E</td>
<td>Inverse hyperbolic tangent function.</td>
</tr>
<tr>
<td>ATOMIC_ADD</td>
<td>(ATOM, VALUE [, STAT])</td>
<td>A</td>
<td>Atomic addition.</td>
</tr>
<tr>
<td>ATOMIC_AND</td>
<td>(ATOM, VALUE [, STAT])</td>
<td>A</td>
<td>Atomic bitwise AND.</td>
</tr>
<tr>
<td>ATOMIC_COMPARE</td>
<td>(ATOM, OLD, COMPARE, VALUE [, STAT])</td>
<td>A</td>
<td>Atomic compare and swap.</td>
</tr>
<tr>
<td>ATOMIC_DEFINE</td>
<td>(ATOM, VALUE [, STAT])</td>
<td>A</td>
<td>Define a variable atomically.</td>
</tr>
<tr>
<td>ATOMIC_FETCH_ADD</td>
<td>(ATOM, VALUE, OLD [, STAT])</td>
<td>A</td>
<td>Atomic fetch and add.</td>
</tr>
<tr>
<td>ATOMIC_FETCH_AND</td>
<td>(ATOM, VALUE, OLD [, STAT])</td>
<td>A</td>
<td>Atomic fetch and bitwise AND.</td>
</tr>
<tr>
<td>ATOMIC_FETCH_OR</td>
<td>(ATOM, VALUE, OLD [, STAT])</td>
<td>A</td>
<td>Atomic fetch and bitwise OR.</td>
</tr>
<tr>
<td>ATOMIC_FETCH_XOR</td>
<td>(ATOM, VALUE, OLD [, STAT])</td>
<td>A</td>
<td>Atomic fetch and bitwise exclusive OR.</td>
</tr>
<tr>
<td>ATOMIC_OR</td>
<td>(ATOM, VALUE [, STAT])</td>
<td>A</td>
<td>Atomic bitwise OR.</td>
</tr>
<tr>
<td>ATOMIC_REF</td>
<td>(VALUE, ATOM [, STAT])</td>
<td>A</td>
<td>Reference a variable atomically.</td>
</tr>
<tr>
<td>ATOMIC_XOR</td>
<td>(ATOM, VALUE [, STAT])</td>
<td>A</td>
<td>Atomic bitwise exclusive OR.</td>
</tr>
<tr>
<td>BESSEL_J0</td>
<td>(X)</td>
<td>E</td>
<td>Bessel function of the 1st kind, order 0.</td>
</tr>
<tr>
<td>BESSEL_J1</td>
<td>(X)</td>
<td>E</td>
<td>Bessel function of the 1st kind, order 1.</td>
</tr>
<tr>
<td>BESSEL_JN</td>
<td>(N, X)</td>
<td>E</td>
<td>Bessel function of the 1st kind, order N.</td>
</tr>
<tr>
<td>BESSEL_Y0</td>
<td>(X)</td>
<td>E</td>
<td>Bessel function of the 2nd kind, order 0.</td>
</tr>
<tr>
<td>BESSEL_Y1</td>
<td>(X)</td>
<td>E</td>
<td>Bessel function of the 2nd kind, order 1.</td>
</tr>
<tr>
<td>BESSEL_YN</td>
<td>(N, X)</td>
<td>E</td>
<td>Bessel function of the 2nd kind, order N.</td>
</tr>
<tr>
<td>BGE</td>
<td>(I, J)</td>
<td>E</td>
<td>Bitwise greater than or equal to.</td>
</tr>
<tr>
<td>BGT</td>
<td>(I, J)</td>
<td>E</td>
<td>Bitwise greater than.</td>
</tr>
<tr>
<td>BLE</td>
<td>(I, J)</td>
<td>E</td>
<td>Bitwise less than or equal to.</td>
</tr>
<tr>
<td>BLT</td>
<td>(I, J)</td>
<td>E</td>
<td>Bitwise less than.</td>
</tr>
<tr>
<td>BTEST</td>
<td>(I, POS)</td>
<td>E</td>
<td>Test single bit in an integer.</td>
</tr>
<tr>
<td>CEILING</td>
<td>(A [, KIND])</td>
<td>E</td>
<td>Least integer greater than or equal to A.</td>
</tr>
<tr>
<td>CHAR</td>
<td>(I [, KIND])</td>
<td>E</td>
<td>Character from code value.</td>
</tr>
<tr>
<td>CMPLX</td>
<td>(X [, KIND]) or (X [, Y, KIND])</td>
<td>E</td>
<td>Conversion to complex type.</td>
</tr>
<tr>
<td>CO_BROADCAST</td>
<td>(A, SOURCE_IMAGE [, STAT, ERRMSG])</td>
<td>C</td>
<td>Broadcast value to images.</td>
</tr>
<tr>
<td>CO_MAX</td>
<td>(A [, RESULT_IMAGE, STAT, ERRMSG])</td>
<td>C</td>
<td>Compute maximum value across images.</td>
</tr>
<tr>
<td>CO_MIN</td>
<td>(A [, RESULT_IMAGE, STAT, ERRMSG])</td>
<td>C</td>
<td>Compute minimum value across images.</td>
</tr>
<tr>
<td>CO_REDUCE</td>
<td>(A, OPERATOR [, RESULT_IMAGE, STAT, ERRMSG])</td>
<td>C</td>
<td>Generalized reduction across images.</td>
</tr>
</tbody>
</table>
Table 16.1: Standard generic intrinsic procedure summary (cont.)

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Arguments</th>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CO_SUM</td>
<td>(A [, RESULT_IMAGE, STAT, ERRMSG])</td>
<td>C</td>
<td>Compute sum across images.</td>
</tr>
<tr>
<td>COMMAND_ARGUMENT_COUNT</td>
<td>( )</td>
<td>T</td>
<td>Number of command arguments.</td>
</tr>
<tr>
<td>CONJG</td>
<td>(Z)</td>
<td>E</td>
<td>Conjugate of a complex number.</td>
</tr>
<tr>
<td>COS</td>
<td>(X)</td>
<td>E</td>
<td>Cosine function.</td>
</tr>
<tr>
<td>COSH</td>
<td>(X)</td>
<td>E</td>
<td>Hyperbolic cosine function.</td>
</tr>
<tr>
<td>COSHAPE</td>
<td>(COARRAY [, KIND])</td>
<td>I</td>
<td>Sizes of codimensions of a coarray.</td>
</tr>
<tr>
<td>COUNT</td>
<td>(MASK [, DIM, KIND])</td>
<td>T</td>
<td>Logical array reduced by counting true values.</td>
</tr>
<tr>
<td>CPU_TIME</td>
<td>(TIME)</td>
<td>S</td>
<td>Processor time used.</td>
</tr>
<tr>
<td>CSHIFT</td>
<td>(ARRAY, SHIFT [, DIM])</td>
<td>T</td>
<td>Circular shift of an array.</td>
</tr>
<tr>
<td>DATE_AND_TIME</td>
<td>([DATE, TIME, ZONE, VALUES])</td>
<td>S</td>
<td>Date and time.</td>
</tr>
<tr>
<td>DBLE</td>
<td>(A)</td>
<td>E</td>
<td>Conversion to double precision real.</td>
</tr>
<tr>
<td>DIGITS</td>
<td>(X)</td>
<td>I</td>
<td>Significant digits in numeric model.</td>
</tr>
<tr>
<td>DIM</td>
<td>(X, Y)</td>
<td>E</td>
<td>Maximum of X − Y and zero.</td>
</tr>
<tr>
<td>DOT_PRODUCT</td>
<td>(VECTOR_A, VECTOR_B)</td>
<td>T</td>
<td>Dot product of two vectors.</td>
</tr>
<tr>
<td>DPROD</td>
<td>(X, Y)</td>
<td>E</td>
<td>Double precision real product.</td>
</tr>
<tr>
<td>DSHIFTLT</td>
<td>(I, J, SHIFT)</td>
<td>E</td>
<td>Combined left shift.</td>
</tr>
<tr>
<td>DSHIFTTR</td>
<td>(I, J, SHIFT)</td>
<td>E</td>
<td>Combined right shift.</td>
</tr>
<tr>
<td>EOSHIFT</td>
<td>(ARRAY, SHIFT [, BOUNDARY, DIM])</td>
<td>T</td>
<td>End-off shift of the elements of an array.</td>
</tr>
<tr>
<td>EPSILON</td>
<td>(X)</td>
<td>I</td>
<td>Model number that is small compared to 1.</td>
</tr>
<tr>
<td>ERF</td>
<td>(X)</td>
<td>E</td>
<td>Error function.</td>
</tr>
<tr>
<td>ERFC</td>
<td>(X)</td>
<td>E</td>
<td>Complementary error function.</td>
</tr>
<tr>
<td>EVENT_QUERY</td>
<td>(EVENT, COUNT [, STAT])</td>
<td>S</td>
<td>Query event count.</td>
</tr>
<tr>
<td>EXECUTE_COMMAND_LINE</td>
<td>(COMMAND [, WAIT, CMDSTAT, CMDMSG])</td>
<td>S</td>
<td>Execute a command line.</td>
</tr>
<tr>
<td>EXP</td>
<td>(X)</td>
<td>E</td>
<td>Exponential function.</td>
</tr>
<tr>
<td>EXPONENT</td>
<td>(X)</td>
<td>E</td>
<td>Exponent of floating-point number.</td>
</tr>
<tr>
<td>EXTENDS_TYPE_OF</td>
<td>(A, MOLD)</td>
<td>I</td>
<td>Dynamic type extension inquiry.</td>
</tr>
<tr>
<td>FAILED_IMAGES</td>
<td>([TEAM, KIND])</td>
<td>T</td>
<td>Indices of failed images.</td>
</tr>
<tr>
<td>FINDLOC</td>
<td>(ARRAY, VALUE, DIM [, MASK, KIND, BACK]) or (ARRAY, VALUE [, MASK, KIND, BACK])</td>
<td>T</td>
<td>Location(s) of a specified value.</td>
</tr>
<tr>
<td>FLOOR</td>
<td>(A [, KIND])</td>
<td>E</td>
<td>Greatest integer less than or equal to A.</td>
</tr>
<tr>
<td>FRACTION</td>
<td>(X)</td>
<td>E</td>
<td>Fractional part of number.</td>
</tr>
<tr>
<td>GAMMA</td>
<td>(X)</td>
<td>E</td>
<td>Gamma function.</td>
</tr>
<tr>
<td>GET_COMMAND</td>
<td>([COMMAND, LENGTH, STATUS, ERRMSG])</td>
<td>S</td>
<td>Get program invocation command.</td>
</tr>
<tr>
<td>GET_COMMAND_ARGUMENT</td>
<td>(NUMBER [, VALUE, LENGTH, STATUS, ERRMSG])</td>
<td>S</td>
<td>Get program invocation argument.</td>
</tr>
<tr>
<td>GET_ENVIRONMENT_VARIABLE</td>
<td>(NAME [, VALUE, LENGTH, STATUS, TRIM_NAME, ERRMSG])</td>
<td>S</td>
<td>Get environment variable.</td>
</tr>
<tr>
<td>Procedure</td>
<td>Arguments</td>
<td>Class</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------</td>
<td>-------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>GET_TEAM</td>
<td>([LEVEL])</td>
<td>T</td>
<td>Team.</td>
</tr>
<tr>
<td>HUGE</td>
<td>(X)</td>
<td>I</td>
<td>Largest model number.</td>
</tr>
<tr>
<td>HYPOT</td>
<td>(X, Y)</td>
<td>E</td>
<td>Euclidean distance function.</td>
</tr>
<tr>
<td>IACHAR</td>
<td>(C [, KIND])</td>
<td>E</td>
<td>ASCII code value for character.</td>
</tr>
<tr>
<td>IALL</td>
<td>(ARRAY, DIM [, MASK]) or (ARRAY [, MASK])</td>
<td>T</td>
<td>Array reduced by IAND function.</td>
</tr>
<tr>
<td>IAND</td>
<td>(I, J)</td>
<td>E</td>
<td>Bitwise AND.</td>
</tr>
<tr>
<td>IANY</td>
<td>(ARRAY, DIM [, MASK]) or (ARRAY [, MASK])</td>
<td>T</td>
<td>Array reduced by IOR function.</td>
</tr>
<tr>
<td>IBCLR</td>
<td>(I, POS)</td>
<td>E</td>
<td>I with bit POS replaced by zero.</td>
</tr>
<tr>
<td>IBITS</td>
<td>(I, POS, LEN)</td>
<td>E</td>
<td>Specified sequence of bits.</td>
</tr>
<tr>
<td>IBSET</td>
<td>(I, POS)</td>
<td>E</td>
<td>I with bit POS replaced by one.</td>
</tr>
<tr>
<td>ICHAR</td>
<td>(C [, KIND])</td>
<td>E</td>
<td>Code value for character.</td>
</tr>
<tr>
<td>IEOR</td>
<td>(I, J)</td>
<td>E</td>
<td>Bitwise exclusive OR.</td>
</tr>
<tr>
<td>IMAGE_INDEX</td>
<td>(COARRAY, SUB) or (COARRAY, SUB, TEAM) or (COARRAY, SUB, TEAM_NUMBER)</td>
<td>I</td>
<td>Image index from cosubscripts.</td>
</tr>
<tr>
<td>IMAGE_STATUS</td>
<td>(IMAGE [, TEAM])</td>
<td>T</td>
<td>Image execution state.</td>
</tr>
<tr>
<td>INDEX</td>
<td>(STRING, SUBSTRING [, BACK, KIND])</td>
<td>E</td>
<td>Character string search.</td>
</tr>
<tr>
<td>INT</td>
<td>(A [, KIND])</td>
<td>E</td>
<td>Conversion to integer type.</td>
</tr>
<tr>
<td>IOR</td>
<td>(I, J)</td>
<td>E</td>
<td>Bitwise inclusive OR.</td>
</tr>
<tr>
<td>IPARITY</td>
<td>(ARRAY, DIM [, MASK]) or (ARRAY [, MASK])</td>
<td>T</td>
<td>Array reduced by IEOR function.</td>
</tr>
<tr>
<td>ISHFT</td>
<td>(I, SHIFT)</td>
<td>E</td>
<td>Logical shift.</td>
</tr>
<tr>
<td>ISHTFC</td>
<td>(I, SHIFT [, SIZE])</td>
<td>E</td>
<td>Circular shift of the rightmost bits.</td>
</tr>
<tr>
<td>IS_CONTIGUOUS</td>
<td>(ARRAY)</td>
<td>I</td>
<td>Array contiguity test (8.5.7).</td>
</tr>
<tr>
<td>IS_IOSTAT_END</td>
<td>(I)</td>
<td>E</td>
<td>IOSTAT value test for end of file.</td>
</tr>
<tr>
<td>IS_IOSTAT_EOR</td>
<td>(I)</td>
<td>E</td>
<td>IOSTAT value test for end of record.</td>
</tr>
<tr>
<td>KIND</td>
<td>(X)</td>
<td>I</td>
<td>Value of the kind type parameter of X.</td>
</tr>
<tr>
<td>LBOUND</td>
<td>(ARRAY [, DIM, KIND])</td>
<td>I</td>
<td>Lower bound(s).</td>
</tr>
<tr>
<td>LCOBOUND</td>
<td>(COARRAY [, DIM, KIND])</td>
<td>I</td>
<td>Lower cobound(s) of a coarray.</td>
</tr>
<tr>
<td>LEADZ</td>
<td>(I)</td>
<td>E</td>
<td>Number of leading zero bits.</td>
</tr>
<tr>
<td>LEN</td>
<td>(STRING [, KIND])</td>
<td>I</td>
<td>Length of a character entity.</td>
</tr>
<tr>
<td>LEN_TRIM</td>
<td>(STRING [, KIND])</td>
<td>E</td>
<td>Length without trailing blanks.</td>
</tr>
<tr>
<td>LGE</td>
<td>(STRING_A, STRING_B)</td>
<td>E</td>
<td>ASCII greater than or equal.</td>
</tr>
<tr>
<td>LGT</td>
<td>(STRING_A, STRING_B)</td>
<td>E</td>
<td>ASCII greater than.</td>
</tr>
<tr>
<td>LLE</td>
<td>(STRING_A, STRING_B)</td>
<td>E</td>
<td>ASCII less than or equal.</td>
</tr>
<tr>
<td>LLT</td>
<td>(STRING_A, STRING_B)</td>
<td>E</td>
<td>ASCII less than.</td>
</tr>
<tr>
<td>LOG</td>
<td>(X)</td>
<td>E</td>
<td>Natural logarithm.</td>
</tr>
<tr>
<td>LOG_GAMMA</td>
<td>(X)</td>
<td>E</td>
<td>Logarithm of the absolute value of the gamma function.</td>
</tr>
<tr>
<td>LOG10</td>
<td>(X)</td>
<td>E</td>
<td>Common logarithm.</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>(L [, KIND])</td>
<td>E</td>
<td>Conversion between kinds of logical.</td>
</tr>
<tr>
<td>MASKL</td>
<td>(I [, KIND])</td>
<td>E</td>
<td>Left justified mask.</td>
</tr>
<tr>
<td>MASKR</td>
<td>(I [, KIND])</td>
<td>E</td>
<td>Right justified mask.</td>
</tr>
<tr>
<td>MATMUL</td>
<td>(MATRIX_A, MATRIX_B)</td>
<td>T</td>
<td>Matrix multiplication.</td>
</tr>
<tr>
<td>MAX</td>
<td>(A1, A2 [, A3, ...])</td>
<td>E</td>
<td>Maximum value.</td>
</tr>
<tr>
<td>MAXEXPONENT</td>
<td>(X)</td>
<td>I</td>
<td>Maximum exponent of a real model.</td>
</tr>
<tr>
<td>MAXLOC</td>
<td>(ARRAY, DIM [, MASK, KIND, BACK]) or (ARRAY [, MASK, KIND, BACK])</td>
<td>T</td>
<td>Location(s) of maximum value.</td>
</tr>
</tbody>
</table>
Table 16.1: Standard generic intrinsic procedure summary (cont.)

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Arguments</th>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAXVAL</td>
<td>(ARRAY, DIM [, MASK]) or (ARRAY [, MASK])</td>
<td>T</td>
<td>Maximum value(s) of array.</td>
</tr>
<tr>
<td>MERGE</td>
<td>(TSOURCE, FSOURCE, MASK)</td>
<td>E</td>
<td>Expression value selection.</td>
</tr>
<tr>
<td>MERGE_BITS</td>
<td>(I, J, MASK)</td>
<td>E</td>
<td>Merge of bits under mask.</td>
</tr>
<tr>
<td>MIN</td>
<td>(A1, A2 [, A3, ...])</td>
<td>E</td>
<td>Minimum value.</td>
</tr>
<tr>
<td>MINEXPONENT</td>
<td>(X)</td>
<td>I</td>
<td>Minimum exponent of a real model.</td>
</tr>
<tr>
<td>MINLOC</td>
<td>(ARRAY, DIM [, MASK, KIND, BACK]) or (ARRAY [, MASK, KIND, BACK])</td>
<td>T</td>
<td>Location(s) of minimum value.</td>
</tr>
<tr>
<td>MINVAL</td>
<td>(ARRAY, DIM [, MASK]) or (ARRAY [, MASK])</td>
<td>T</td>
<td>Minimum value(s) of array.</td>
</tr>
<tr>
<td>MOD</td>
<td>(A, P)</td>
<td>E</td>
<td>Remainder function.</td>
</tr>
<tr>
<td>MODULO</td>
<td>(A, P)</td>
<td>E</td>
<td>Modulo function.</td>
</tr>
<tr>
<td>MOVE_ALLOC</td>
<td>(FROM, TO [, STAT, ERRMSG])</td>
<td>PS</td>
<td>Move an allocation.</td>
</tr>
<tr>
<td>MERGE_BITS</td>
<td>(FROM, FROMPOS, LEN, TO, TOPOS)</td>
<td>ES</td>
<td>Copy a sequence of bits.</td>
</tr>
<tr>
<td>NEAREST</td>
<td>(X, S)</td>
<td>E</td>
<td>Adjacent machine number.</td>
</tr>
<tr>
<td>NEW_LINE</td>
<td>(A)</td>
<td>I</td>
<td>Newline character.</td>
</tr>
<tr>
<td>NINT</td>
<td>(A [, KIND])</td>
<td>E</td>
<td>Nearest integer.</td>
</tr>
<tr>
<td>NORM2</td>
<td>(X) or (X, DIM)</td>
<td>T</td>
<td>$L_2$ norm of an array.</td>
</tr>
<tr>
<td>NOT</td>
<td>(I)</td>
<td>E</td>
<td>Bitwise complement.</td>
</tr>
<tr>
<td>NULL</td>
<td>([MOLD])</td>
<td>T</td>
<td>Disassociated pointer or unallocated allocatable entity.</td>
</tr>
<tr>
<td>NUM_IMAGES</td>
<td>( ) or (TEAM) or (TEAM_-NUMBER)</td>
<td>T</td>
<td>Number of images.</td>
</tr>
<tr>
<td>OUT_OF_RANGE</td>
<td>(X, MOLD [, ROUND])</td>
<td>E</td>
<td>Whether a value cannot be converted safely.</td>
</tr>
<tr>
<td>PACK</td>
<td>(ARRAY, MASK [, VECTOR])</td>
<td>T</td>
<td>Array packed into a vector.</td>
</tr>
<tr>
<td>PARITY</td>
<td>(MASK) or (MASK, DIM)</td>
<td>T</td>
<td>Array reduced by .NEQV. operator.</td>
</tr>
<tr>
<td>POPCNT</td>
<td>([VECTOR])</td>
<td>E</td>
<td>Number of one bits.</td>
</tr>
<tr>
<td>POPPAR</td>
<td>(I)</td>
<td>E</td>
<td>Parity expressed as 0 or 1.</td>
</tr>
<tr>
<td>PRECISION</td>
<td>(X)</td>
<td>I</td>
<td>Decimal precision of a real model.</td>
</tr>
<tr>
<td>PRESENT</td>
<td>(A)</td>
<td>I</td>
<td>Presence of optional argument.</td>
</tr>
<tr>
<td>PRODUCT</td>
<td>(ARRAY, DIM [, MASK]) or (ARRAY [, MASK])</td>
<td>T</td>
<td>Array reduced by multiplication.</td>
</tr>
<tr>
<td>RADIX</td>
<td>(X)</td>
<td>I</td>
<td>Base of a numeric model.</td>
</tr>
<tr>
<td>RANDOM_INIT</td>
<td>(REPEATABLE, IMAGE_-DISTINCT)</td>
<td>S</td>
<td>Initialise the pseudorandom number generator.</td>
</tr>
<tr>
<td>RANDOM_NUMBER</td>
<td>(HARVEST)</td>
<td>S</td>
<td>Generate pseudorandom number(s).</td>
</tr>
<tr>
<td>RANDOM_SEED</td>
<td>([SIZE, PUT, GET])</td>
<td>S</td>
<td>Restart or query the pseudorandom number generator.</td>
</tr>
<tr>
<td>RANGE</td>
<td>(X)</td>
<td>I</td>
<td>Decimal exponent range of a numeric model (16.4).</td>
</tr>
<tr>
<td>RANK</td>
<td>(A)</td>
<td>I</td>
<td>Rank of a data object.</td>
</tr>
<tr>
<td>REAL</td>
<td>(A [, KIND])</td>
<td>E</td>
<td>Conversion to real type.</td>
</tr>
<tr>
<td>REDUCE</td>
<td>(ARRAY, OPERATION, DIM [, MASK, IDENTITY, ORDERED]) or (ARRAY, OPERATION [, MASK, IDENTITY, ORDERED])</td>
<td>T</td>
<td>General reduction of array.</td>
</tr>
</tbody>
</table>
## Table 16.1: Standard generic intrinsic procedure summary

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Arguments</th>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REPEAT</td>
<td>(STRING, NCOPIES)</td>
<td>T</td>
<td>Repetitive string concatenation.</td>
</tr>
<tr>
<td>RESHAPE</td>
<td>(SOURCE, SHAPE [, PAD, ORDER])</td>
<td>T</td>
<td>Arbitrary shape array construction.</td>
</tr>
<tr>
<td>RRSPACING</td>
<td>(X)</td>
<td>E</td>
<td>Reciprocal of relative spacing of model numbers.</td>
</tr>
<tr>
<td>SAME_TYPE_AS</td>
<td>(A, B)</td>
<td>I</td>
<td>Dynamic type equality test.</td>
</tr>
<tr>
<td>SCALE</td>
<td>(X, I)</td>
<td>E</td>
<td>Real number scaled by radix power.</td>
</tr>
<tr>
<td>SCAN</td>
<td>(STRING, SET [, BACK, KIND])</td>
<td>E</td>
<td>Character set membership search.</td>
</tr>
<tr>
<td>SELECTED_CHAR_KIND</td>
<td>(NAME)</td>
<td>T</td>
<td>Character kind selection.</td>
</tr>
<tr>
<td>SELECTED_INT_KIND</td>
<td>(R)</td>
<td>T</td>
<td>Integer kind selection.</td>
</tr>
<tr>
<td>SELECTED_REAL_KIND</td>
<td>([P, R, RADIX])</td>
<td>T</td>
<td>Real kind selection.</td>
</tr>
<tr>
<td>SET_EXPONENT</td>
<td>(X, I)</td>
<td>E</td>
<td>Real value with specified exponent.</td>
</tr>
<tr>
<td>SHAPE</td>
<td>(SOURCE [, KIND])</td>
<td>I</td>
<td>Shape of an array or a scalar.</td>
</tr>
<tr>
<td>SHIFTA</td>
<td>(I, SHIFT)</td>
<td>E</td>
<td>Right shift with fill.</td>
</tr>
<tr>
<td>SHIFTL</td>
<td>(I, SHIFT)</td>
<td>E</td>
<td>Left shift.</td>
</tr>
<tr>
<td>SHIFTR</td>
<td>(I, SHIFT)</td>
<td>E</td>
<td>Right shift.</td>
</tr>
<tr>
<td>SIGN</td>
<td>(A, B)</td>
<td>E</td>
<td>Magnitude of A with the sign of B.</td>
</tr>
<tr>
<td>SIN</td>
<td>(X)</td>
<td>E</td>
<td>Sine function.</td>
</tr>
<tr>
<td>SINC</td>
<td>(X)</td>
<td>E</td>
<td>Hyperbolic sine function.</td>
</tr>
<tr>
<td>SIZE</td>
<td>(ARRAY [, DIM, KIND])</td>
<td>I</td>
<td>Size of an array or one extent.</td>
</tr>
<tr>
<td>SPACING</td>
<td>(X)</td>
<td>E</td>
<td>Spacing of model numbers.</td>
</tr>
<tr>
<td>SPREAD</td>
<td>(SOURCE, DIM, NCOPIES)</td>
<td>T</td>
<td>Value replicated in a new dimension.</td>
</tr>
<tr>
<td>SQRT</td>
<td>(X)</td>
<td>E</td>
<td>Square root.</td>
</tr>
<tr>
<td>STOPPED_IMAGES</td>
<td>([TEAM, KIND])</td>
<td>T</td>
<td>Indices of stopped images.</td>
</tr>
<tr>
<td>STORAGE_SIZE</td>
<td>(A [, KIND])</td>
<td>I</td>
<td>Storage size in bits.</td>
</tr>
<tr>
<td>SUM</td>
<td>(ARRAY, DIM [, MASK])</td>
<td>T</td>
<td>Array reduced by addition.</td>
</tr>
<tr>
<td>SYSTEM_CLOCK</td>
<td>((COUNT, COUNT_RATE, COUNT_MAX))</td>
<td>S</td>
<td>Query system clock.</td>
</tr>
<tr>
<td>TAN</td>
<td>(X)</td>
<td>E</td>
<td>Tangent function.</td>
</tr>
<tr>
<td>TANH</td>
<td>(X)</td>
<td>E</td>
<td>Hyperbolic tangent function.</td>
</tr>
<tr>
<td>TEAM_NUMBER</td>
<td>([TEAM])</td>
<td>T</td>
<td>Team number.</td>
</tr>
<tr>
<td>THIS_IMAGE</td>
<td>([TEAM])</td>
<td>T</td>
<td>Index of the invoking image.</td>
</tr>
<tr>
<td>THIS_IMAGE</td>
<td>(COARRAY [, TEAM]) or (COARRAY, DIM [, TEAM])</td>
<td>T</td>
<td>Cosubscript(s) for this image.</td>
</tr>
<tr>
<td>TINY</td>
<td>(X)</td>
<td>I</td>
<td>Smallest positive model number.</td>
</tr>
<tr>
<td>TRAILZ</td>
<td>(I)</td>
<td>E</td>
<td>Number of trailing zero bits.</td>
</tr>
<tr>
<td>TRANSFER</td>
<td>(SOURCE, MOLD [, SIZE])</td>
<td>T</td>
<td>Transfer physical representation.</td>
</tr>
<tr>
<td>TRANSPOSE</td>
<td>(MATRIX)</td>
<td>T</td>
<td>Transpose of an array of rank two.</td>
</tr>
<tr>
<td>TRIM</td>
<td>(STRING)</td>
<td>T</td>
<td>String without trailing blanks.</td>
</tr>
<tr>
<td>UBOUND</td>
<td>(ARRAY [, DIM, KIND])</td>
<td>I</td>
<td>Upper bound(s).</td>
</tr>
<tr>
<td>UCOBOUND</td>
<td>(COARRAY [, DIM, KIND])</td>
<td>I</td>
<td>Upper cobound(s) of a coarray.</td>
</tr>
<tr>
<td>UNPACK</td>
<td>(VECTOR, MASK, FIELD)</td>
<td>T</td>
<td>Vector unpacked into an array.</td>
</tr>
<tr>
<td>VERIFY</td>
<td>(STRING, SET [, BACK, KIND])</td>
<td>E</td>
<td>Character set non-membership search.</td>
</tr>
</tbody>
</table>

---

1. The effect of calling EXECUTE_COMMAND_LINE on any image other than image 1 in the initial team is processor dependent.
If \texttt{RANDOM\_INIT} or \texttt{RANDOM\_SEED} is called in a segment A, and \texttt{RANDOM\_INIT, RANDOM\_SEED, or RANDOM\_NUMBER} is called in segment B, then segments A and B shall be ordered. It is processor dependent whether each image uses a separate random number generator, or if some or all images use common random number generators.

The use of all other standard intrinsic procedures in unordered segments is subject only to their argument use following the rules in 11.6.2.

\section*{16.8 Specific names for standard intrinsic functions}

\begin{enumerate}
\item Except for \texttt{AMAX0, AMIN0, MAX1, and MIN1}, the result type of the specific function is the same that the result type of the corresponding generic function reference would be if it were invoked with the same arguments as the specific function.
\item A function listed in Table 16.3 is not permitted to be used as an actual argument (15.5.1, C1534), as a target in a procedure pointer assignment statement (10.2.2.2, C1030), as an initial target in a procedure declaration statement (15.4.3.6, C1519), or to specify an interface (15.4.3.6, C1515).
\end{enumerate}

\begin{table}[h]
\centering
\caption{Unrestricted specific intrinsic functions}
\begin{tabular}{|l|l|l|}
\hline
Specific name & Generic name & Argument type and kind \\
\hline
\texttt{ABS} & \texttt{ABS} & default real \\
\texttt{ACOS} & \texttt{ACOS} & default real \\
\texttt{AIMAG} & \texttt{AIMAG} & default complex \\
\texttt{AINT} & \texttt{AINT} & default real \\
\texttt{ALOG} & \texttt{LOG} & default real \\
\texttt{ALOG10} & \texttt{LOG10} & default real \\
\texttt{AMOD} & \texttt{MOD} & default real \\
\texttt{ANINT} & \texttt{ANINT} & default real \\
\texttt{ASIN} & \texttt{ASIN} & default real \\
\texttt{ATAN} & \texttt{ATAN (X)} & default real \\
\texttt{ATAN2} & \texttt{ATAN2} & default real \\
\texttt{CABS} & \texttt{ABS} & default complex \\
\texttt{CCOS} & \texttt{COS} & default complex \\
\texttt{CEXP} & \texttt{EXP} & default complex \\
\texttt{CLOG} & \texttt{LOG} & default complex \\
\texttt{CONJG} & \texttt{CONJG} & default complex \\
\texttt{COS} & \texttt{COS} & default real \\
\texttt{COSH} & \texttt{COSH} & default real \\
\texttt{CSIN} & \texttt{SIN} & default complex \\
\texttt{CSQRT} & \texttt{SQR} & default complex \\
\texttt{DABS} & \texttt{ABS} & double precision real \\
\texttt{DACOS} & \texttt{ACOS} & double precision real \\
\texttt{DASIN} & \texttt{ASIN} & double precision real \\
\texttt{DATAN} & \texttt{ATAN} & double precision real \\
\texttt{DATAN2} & \texttt{ATAN2} & double precision real \\
\texttt{DCOS} & \texttt{COS} & double precision real \\
\texttt{DCOSH} & \texttt{COSH} & double precision real \\
\texttt{DDIM} & \texttt{DIM} & double precision real \\
\texttt{DEXP} & \texttt{EXP} & double precision real \\
\texttt{DIM} & \texttt{DIM} & default real \\
\texttt{DINT} & \texttt{AINT} & double precision real \\
\texttt{DLOG} & \texttt{LOG} & double precision real \\
\texttt{DLOG10} & \texttt{LOG10} & double precision real \\
\texttt{DMOD} & \texttt{MOD} & double precision real \\
\texttt{DNINT} & \texttt{ANINT} & double precision real \\
\texttt{DPROD} & \texttt{DPROD} & default real \\
\texttt{DSIGN} & \texttt{SIGN} & double precision real \\
\texttt{DSIN} & \texttt{SIN} & double precision real \\
\texttt{DSINH} & \texttt{SINH} & double precision real \\
\texttt{DSQRT} & \texttt{SQR} & double precision real \\
\texttt{DTAN} & \texttt{TAN} & double precision real \\
\texttt{DTANH} & \texttt{TANH} & double precision real \\
\texttt{EXP} & \texttt{EXP} & default real \\
\texttt{IABS} & \texttt{ABS} & default integer \\
\texttt{IDIM} & \texttt{DIM} & default integer \\
\texttt{IDNINT} & \texttt{NINT} & double precision real \\
\texttt{INDEX} & \texttt{INDEX} & default character \\
\hline
\end{tabular}
\end{table}
### Unrestricted specific intrinsic functions (cont.)

<table>
<thead>
<tr>
<th>Specific name</th>
<th>Generic name</th>
<th>Argument type and kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISIGN</td>
<td>SIGN</td>
<td>default integer</td>
</tr>
<tr>
<td>LEN</td>
<td>LEN</td>
<td>default character</td>
</tr>
<tr>
<td>MOD</td>
<td>MOD</td>
<td>default integer</td>
</tr>
<tr>
<td>NINT</td>
<td>NINT</td>
<td>default real</td>
</tr>
<tr>
<td>SIGN</td>
<td>SIGN</td>
<td>default real</td>
</tr>
<tr>
<td>SIN</td>
<td>SIN</td>
<td>default real</td>
</tr>
<tr>
<td>SINH</td>
<td>SINH</td>
<td>default real</td>
</tr>
<tr>
<td>SQRT</td>
<td>SQRT</td>
<td>default real</td>
</tr>
<tr>
<td>TAN</td>
<td>TAN</td>
<td>default real</td>
</tr>
<tr>
<td>TANH</td>
<td>TANH</td>
<td>default real</td>
</tr>
</tbody>
</table>

### Restricted specific intrinsic functions

<table>
<thead>
<tr>
<th>Specific name</th>
<th>Generic name</th>
<th>Argument type and kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMAX0 (...)</td>
<td>REAL (MAX (...))</td>
<td>default integer</td>
</tr>
<tr>
<td>AMAX1</td>
<td>MAX</td>
<td>default real</td>
</tr>
<tr>
<td>AMIN0 (...)</td>
<td>REAL (MIN (...))</td>
<td>default integer</td>
</tr>
<tr>
<td>AMIN1</td>
<td>MIN</td>
<td>default real</td>
</tr>
<tr>
<td>CHAR</td>
<td>CHAR</td>
<td>default integer</td>
</tr>
<tr>
<td>DMAX1</td>
<td>MAX</td>
<td>double precision real</td>
</tr>
<tr>
<td>DMIN1</td>
<td>MIN</td>
<td>double precision real</td>
</tr>
<tr>
<td>FLOAT</td>
<td>REAL</td>
<td>default integer</td>
</tr>
<tr>
<td>ICHAR</td>
<td>ICHAR</td>
<td>default character</td>
</tr>
<tr>
<td>IDINT</td>
<td>INT</td>
<td>double precision real</td>
</tr>
<tr>
<td>IFIX</td>
<td>INT</td>
<td>default real</td>
</tr>
<tr>
<td>INT</td>
<td>INT</td>
<td>default real</td>
</tr>
<tr>
<td>LGE</td>
<td>LGE</td>
<td>default character</td>
</tr>
<tr>
<td>LGT</td>
<td>LGT</td>
<td>default character</td>
</tr>
<tr>
<td>LLE</td>
<td>LLE</td>
<td>default character</td>
</tr>
<tr>
<td>LLLT</td>
<td>LLT</td>
<td>default character</td>
</tr>
<tr>
<td>MAX0</td>
<td>MAX</td>
<td>default integer</td>
</tr>
<tr>
<td>MAX1 (...)</td>
<td>INT (MAX (...))</td>
<td>default real</td>
</tr>
<tr>
<td>MIN0</td>
<td>MIN</td>
<td>default integer</td>
</tr>
<tr>
<td>MIN1 (...)</td>
<td>INT (MIN (...))</td>
<td>default real</td>
</tr>
<tr>
<td>REAL</td>
<td>REAL</td>
<td>default integer</td>
</tr>
<tr>
<td>SNGL</td>
<td>REAL</td>
<td>double precision real</td>
</tr>
</tbody>
</table>

#### 16.9 Specifications of the standard intrinsic procedures

##### 16.9.1 General

Detailed specifications of the standard generic intrinsic procedures are provided in 16.9 in alphabetical order.

The types and type parameters of standard intrinsic procedure arguments and function results are determined by these specifications. The “Argument(s)” paragraphs specify requirements on the actual arguments of the procedures. The result characteristics are sometimes specified in terms of the characteristics of dummy arguments. A program shall not invoke an intrinsic procedure under circumstances where a value to be assigned to a subroutine argument or returned as a function result is not representable by objects of the specified type and type parameters.

If an IEEE infinity is assigned or returned by an intrinsic procedure, the intrinsic module IEEE_ARITHMETIC is accessible, and the actual arguments were finite numbers, the flag IEEE_OVERFLOW or IEEE_DIVIDE_-BY_ZERO shall signal. If an IEEE NaN is assigned or returned, the actual arguments were finite numbers, the intrinsic module IEEE_ARITHMETIC is accessible, and the exception IEEE_INVALID is supported, the flag IEEE_INVALID shall signal. If no IEEE infinity or NaN is assigned or returned, these flags shall have the same status as when the intrinsic procedure was invoked.
16.9.2 ABS (A)

1 Description. Absolute value.

2 Class. Elemental function.

3 Argument. A shall be of type integer, real, or complex.

4 Result Characteristics. The same as A except that if A is complex, the result is real.

5 Result Value. If A is of type integer or real, the value of the result is |A|; if A is complex with value (x, y), the result is equal to a processor-dependent approximation to $\sqrt{x^2 + y^2}$ computed without undue overflow or underflow.

6 Example. ABS ((3.0, 4.0)) has the value 5.0 (approximately).

16.9.3 ACHAR (I [, KIND])

1 Description. Character from ASCII code value.

2 Class. Elemental function.

3 Arguments.

4 I shall be of type integer.

5 KIND (optional) shall be a scalar integer constant expression.

6 Result Characteristics. Character of length one. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default character.

7 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 the processor is capable of representing that character in the character kind of the result; otherwise, the result is processor dependent. ACHAR (IACHAR (C)) shall have the value C for any character C capable of representation as a default character.

8 Example. ACHAR (88) has the value ‘X’.

16.9.4 ACOS (X)

1 Description. Arccosine (inverse cosine) function.

2 Class. Elemental function.

3 Argument. X shall be of type real with a value that satisfies the inequality $|X| \leq 1$, or of type complex.

4 Result Characteristics. Same as X.

5 Result Value. The result has a value equal to a processor-dependent approximation to arccos(X). If it is real it is expressed in radians and lies in the range $0 \leq \text{ACOS} (X) \leq \pi$. If it is complex the real part is expressed in radians and lies in the range $0 \leq \text{REAL} (\text{ACOS} (X)) \leq \pi$.

6 Example. ACOS (0.54030231) has the value 1.0 (approximately).

16.9.5 ACOSH (X)

1 Description. Inverse hyperbolic cosine function.

2 Class. Elemental function.

3 Argument. X shall be of type real or complex.
Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to the inverse hyperbolic cosine function of X. If the result is complex the imaginary part is expressed in radians and lies in the range

\[ 0 \leq \text{AIMAG (ACOSH (X))} \leq \pi \]

Example. ACOSH (1.5430806) has the value 1.0 (approximately).

16.9.6 ADJUSTL (STRING)

Description. Left-justified string value.

Class. Elemental function.

Argument. STRING shall be of type character.

Result Characteristics. Character of the same length and kind type parameter as STRING.

Result Value. The value of the result is the same as STRING except that any leading blanks have been deleted and the same number of trailing blanks have been inserted.

Example. ADJUSTL ('WORD') has the value 'WORD'.

16.9.7 ADJUSTR (STRING)

Description. Right-justified string value.

Class. Elemental function.

Argument. STRING shall be of type character.

Result Characteristics. 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.

Example. ADJUSTR ('WORD ') has the value 'WORD'.

16.9.8 AIMAG (Z)

Description. Imaginary part of a complex number.

Class. Elemental function.

Argument. Z shall be of type complex.

Result Characteristics. Real with the same kind type parameter as Z.

Result Value. If Z has the value \((x, y)\), the result has the value \(y\).

Example. AIMAG ((2.0, 3.0)) has the value 3.0.

16.9.9 AINT (A [, KIND])

Description. Truncation toward 0 to a whole number.

Class. Elemental function.

Arguments.

A shall be of type real.
KIND (optional) shall be a scalar integer constant expression.

Result Characteristics. The result is of type real. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of A.

Result Value. If \(|A| < 1\), AINT (A) has the value 0; if \(|A| \geq 1\), AINT (A) 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 (2.783) has the value 2.0. AINT (−2.783) has the value −2.0.

16.9.10 ALL (MASK) or ALL (MASK, DIM)

Description. Array reduced by .AND. operator.

Class. Transformational function.

Arguments.

MASK shall be a logical array.

DIM shall be an integer scalar with value in the range 1 ≤ DIM ≤ n, where n is the rank of MASK.

Result Characteristics. The result is of type logical with the same kind type parameter as MASK. It is scalar if DIM does not appear or \(n = 1\); otherwise, the result has rank \(n - 1\) and shape \([d_1, d_2, \ldots, d_{DIM-1}, d_{DIM+1}, \ldots, d_n]\) where \([d_1, d_2, \ldots, d_n]\) is the shape of MASK.

Result Value.

Case (i): The result of ALL (MASK) has the value true if all elements of MASK are true or if MASK has size zero, and the result has value false if any element of MASK is false.

Case (ii): If MASK has rank one, ALL (MASK, DIM) is equal to ALL (MASK). Otherwise, the value of element \((s_1, s_2, \ldots, s_{DIM-1}, s_{DIM+1}, \ldots, s_n)\) of ALL (MASK, DIM) is equal to ALL (MASK \((s_1, s_2, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n))\).

Examples.

Case (i): The value of ALL ([.TRUE., .FALSE., .TRUE.]) is false.

Case (ii): If B is the array \[
\begin{bmatrix}
1 & 3 & 5 \\
2 & 4 & 6 \\
\end{bmatrix}
\]
and C is the array \[
\begin{bmatrix}
0 & 3 & 5 \\
7 & 4 & 8 \\
\end{bmatrix}
\]
then ALL (B /= C, DIM = 1) is [true, false, false] and ALL (B /= C, DIM = 2) is [false, false].

16.9.11 ALLOCATED (ARRAY) or ALLOCATED (SCALAR)

Description. Allocation status of allocatable variable.

Class. Inquiry function.

Arguments.

ARRAY shall be an allocatable array.

SCALAR shall be an allocatable scalar.

Result Characteristics. Default logical scalar.

Result Value. The result has the value true if the argument (ARRAY or SCALAR) is allocated and has the value false if the argument is unallocated.
16.9.12 ANINT (A [, KIND])

Description. Nearest whole number.

Class. Elemental function.

Arguments.
A shall be of type real.
KIND (optional) shall be a scalar integer constant expression.

Result Characteristics. The result is of type real. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of A.

Result Value. The result is the integer nearest A, or if there are two integers equally near A, the result is whichever such integer has the greater magnitude.

Examples. ANINT (2.783) has the value 3.0. ANINT (−2.783) has the value −3.0.

16.9.13 ANY (MASK) or ANY (MASK, DIM)

Description. Array reduced by .OR. operator.

Class. Transformational function.

Arguments.
MASK shall a logical array.
DIM shall be an integer scalar with a value in the range 1 ≤ DIM ≤ n, where n is the rank of MASK.

Result Characteristics. The result is of type logical with the same kind type parameter as MASK. It is scalar if DIM does not appear or n = 1; otherwise, the result has rank n − 1 and shape [d1, d2, ..., dDIM−1, dDIM+1, ..., dn] where [d1, d2, ..., dn] is the shape of MASK.

Result Value.
Case (i): The result of ANY (MASK) has the value true if any element of MASK is true and has the value false if no elements are true or if MASK has size zero.
Case (ii): If MASK has rank one, ANY (MASK, DIM) is equal to ANY (MASK). Otherwise, the value of element (s1, s2, ..., sDIM−1, sDIM+1, ..., sn) of ANY (MASK, DIM) is equal to ANY (MASK (s1, s2, ..., sDIM−1, sDIM+1, ..., sn)).

Examples.
Case (i): The value of ANY ([.TRUE., .FALSE., .TRUE.]) is true.
Case (ii): If B is the array \[
\begin{bmatrix}
1 & 3 & 5 \\
2 & 4 & 6
\end{bmatrix}
\] and C is the array \[
\begin{bmatrix}
0 & 3 & 5 \\
7 & 4 & 8
\end{bmatrix}
\] then ANY (B /= C, DIM = 1) is [true, false, true] and ANY (B /= C, DIM = 2) is [true, true].

16.9.14 ASIN (X)

Description. Arcsine (inverse sine) function.

Class. Elemental function.

Argument. X shall be of type real with a value that satisfies the inequality |X| ≤ 1, or of type complex.

Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to arcsin(X). If it is real it is expressed in radians and lies in the range −π/2 ≤ ASIN (X) ≤ π/2. If it is complex the real part is expressed in radians and lies in the range −π/2 ≤ REAL (ASIN (X)) ≤ π/2.
Example. ASIN (0.84147098) has the value 1.0 (approximately).

16.9.15 ASINH (X)

1 Description. Inverse hyperbolic sine function.

2 Class. Elemental function.

3 Argument. X shall be of type real or complex.

4 Result Characteristics. Same as X.

5 Result Value. The result has a value equal to a processor-dependent approximation to the inverse hyperbolic sine function of X. If the result is complex the imaginary part is expressed in radians and lies in the range $-\pi/2 \leq \text{AIMAG (ASINH (X))} \leq \pi/2$.

Example. ASINH (1.1752012) has the value 1.0 (approximately).

16.9.16 ASSOCIATED (POINTER [, TARGET])

1 Description. Pointer association status inquiry.

2 Class. Inquiry function.

3 Arguments.

POINTER shall be a pointer. It may be of any type or may be a procedure pointer. Its pointer association status shall not be undefined.

TARGET (optional) shall be allowable as the data-target or proc-target in a pointer assignment statement (10.2.2) in which POINTER is data-pointer-object or proc-pointer-object. If TARGET is a pointer then its pointer association status shall not be undefined.

4 Result Characteristics. Default logical scalar.

5 Result Value.

Case (i): If TARGET is absent, the result is true if and only if POINTER is associated with a target.

Case (ii): If TARGET is present and is a procedure, the result is true if and only if POINTER is associated with TARGET and, if TARGET is an internal procedure, they have the same host instance.

Case (iii): If TARGET is present and is a procedure pointer, the result is true if and only if POINTER and TARGET are associated with the same procedure and, if the procedure is a procedure pointer, they have the same host instance.

Case (iv): If TARGET is present and is a scalar target, the result is true if and only if TARGET is not a zero-sized storage sequence and POINTER is associated with a target that occupies the same storage units as TARGET.

Case (v): If TARGET is present and is an array target, the result is true if and only if POINTER is associated with a target that has the same shape as TARGET, is neither of size zero nor an array whose elements are zero-sized storage sequences, and occupies the same storage units as TARGET in array element order.

Case (vi): If TARGET is present and is a scalar pointer, the result is true if and only if POINTER and TARGET are both associated, have the same shape, are neither of size zero nor arrays whose elements are zero-sized storage sequences, and occupy the same storage units in array element order.
NOTE 16.10

The references to TARGET in the above cases are referring to properties that might be possessed by the actual argument, so the case of TARGET being a disassociated pointer will be covered by case (iii), (vi), or (vii).

6 Examples. ASSOCIATED (CURRENT, HEAD) is true if CURRENT is associated with the target HEAD.

After the execution of

\[ \text{A\_PART = A (:N)} \]

ASSOCIATED (A\_PART, A) is true if N is equal to UBOUND (A, DIM = 1). After the execution of

\[ \text{NULLIFY \ (CUR); NULLIFY \ (TOP)} \]

ASSOCIATED (CUR, TOP) is false.

16.9.17 ATAN (X) or ATAN (Y, X)

1 Description. Arctangent (inverse tangent) function.

2 Class. Elemental function.

3 Arguments.

Y shall be of type real.

X If Y appears, X shall be of type real with the same kind type parameter as Y. If Y has the value zero, X shall not have the value zero. If Y does not appear, X shall be of type real or complex.

4 Result Characteristics. Same as X.

5 Result Value. If Y appears, the result is the same as the result of ATAN2 (Y, X). If Y does not appear, the result has a value equal to a processor-dependent approximation to \( \arctan(X) \) whose real part is expressed in radians and lies in the range \(-\pi/2 \leq ATAN (X) \leq \pi/2\).

6 Example. ATAN (1.5574077) has the value 1.0 (approximately).

16.9.18 ATAN2 (Y, X)

1 Description. Arctangent (inverse tangent) function.

2 Class. Elemental function.

3 Arguments.

Y shall be of type real.

X shall be of the same type and kind type parameter as Y. If Y has the value zero, X shall not have the value zero.

4 Result Characteristics. Same as X.

5 Result Value. The result has a value equal to a processor-dependent approximation to the principal value of the argument of the complex number \( (X, Y) \), expressed in radians. It lies in the range \(-\pi \leq ATAN2 (Y, X) \leq \pi \) and is equal to a processor-dependent approximation to a value of \( \arctan(Y/X) \) if \( X \neq 0 \). If \( Y > 0 \), the result is positive. If \( Y = 0 \) and \( X > 0 \), the result is \( Y \). If \( Y = 0 \) and \( X < 0 \), then the result is approximately \( \pi \) if \( Y \) is positive real zero or the processor does not distinguish between positive and negative real zero, and approximately \(-\pi \) if \( Y \) is negative real zero. If \( Y < 0 \), the result is negative. If \( X = 0 \), the absolute value of the result is approximately \( \pi/2 \).

6 Examples. ATAN2 (1.5574077, 1.0) has the value 1.0 (approximately). If \( Y \) has the value \( \begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix} \) and \( X \) has the value \( \begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix} \), the value of ATAN2 (Y, X) is approximately \( \begin{bmatrix} 3\pi/4 & \pi/4 \\ -3\pi/4 & -\pi/4 \end{bmatrix} \).
16.9.19  ATANH (X)

Description. Inverse hyperbolic tangent function.

Class. Elemental function.

Argument. X shall be of type real or complex.

Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to the inverse hyperbolic tangent function of X. If the result is complex the imaginary part is expressed in radians and lies in the range $-\pi/2 \leq \text{AIMAG (ATANH (X))} \leq \pi/2$.

Example. ATANH (0.76159416) has the value 1.0 (approximately).

16.9.20  ATOMIC_ADD (ATOM, VALUE [, STAT])

Description. Atomic addition.

Class. Atomic subroutine.

Arguments.

ATOM shall be a scalar coarray or coindexed object. It shall be of type integer with kind ATOMIC_INT_KIND from the intrinsic module ISO_FORTRAN_ENV. It is an INTENT (INOUT) argument. If an error condition occurs, ATOM becomes undefined; otherwise, it becomes defined with the value of ATOM + VALUE.

VALUE shall be an integer scalar. It is an INTENT (IN) argument. The values of VALUE and ATOM + VALUE shall be representable in ATOMIC_INT_KIND.

STAT (optional) shall be a noncoindexed integer scalar with a decimal range of at least four. It is an INTENT (OUT) argument. It is assigned a value as specified in 16.5. If an error condition occurs and STAT is not present, error termination is initiated.

Example. CALL ATOMIC_ADD (I [3], 42) will cause I on image 3 to become defined with the value 46 if the value of I [3] is 4 when the atomic operation is executed.

16.9.21  ATOMIC_AND (ATOM, VALUE [, STAT])

Description. Atomic bitwise AND.

Class. Atomic subroutine.

Arguments.

ATOM shall be a scalar coarray or coindexed object. It shall be of type integer with kind ATOMIC_INT_KIND from the intrinsic module ISO_FORTRAN_ENV. It is an INTENT (INOUT) argument. If an error condition occurs, ATOM becomes undefined; otherwise, it becomes defined with the value of IAND (ATOM, INT (VALUE, ATOMIC_INT_KIND)).

VALUE shall be an integer scalar. It is an INTENT (IN) argument. The value of VALUE shall be representable in ATOMIC_INT_KIND.

STAT (optional) shall be a noncoindexed integer scalar with a decimal range of at least four. It is an INTENT (OUT) argument. It is assigned a value as specified in 16.5. If an error condition occurs and STAT is not present, error termination is initiated.

Example. CALL ATOMIC_AND (I [3], 6) will cause I on image 3 to become defined with the value 4 if the value of I [3] is 5 when the atomic operation is executed.
16.9.22 ATOMIC_CAS (ATOM, OLD, COMPARE, NEW [, STAT])

1 Description. Atomic compare and swap.

2 Class. Atomic subroutine.

3 Arguments.

ATOM shall be a scalar coarray or coindexed object. It shall be of type integer with kind ATOMIC_INT_KIND from the intrinsic module ISO_FORTRAN_ENV, or of type logical with kind ATOMIC_LOGICAL_KIND from the intrinsic module ISO_FORTRAN_ENV. It is an INTENT (INOUT) argument. If an error condition occurs, ATOM becomes undefined; otherwise, if ATOM is of type integer and equal to COMPARE, or of type logical and equivalent to COMPARE, it becomes defined with the value of NEW.

OLD shall be scalar and of the same type and kind as ATOM. It is an INTENT (OUT) argument. If an error condition occurs, it becomes undefined; otherwise, it becomes defined with the value that ATOM had at the start of the atomic operation.

COMPARE shall be scalar and of the same type and kind as ATOM. It is an INTENT (IN) argument.

NEW shall be scalar and of the same type and kind as ATOM. It is an INTENT (IN) argument.

STAT (optional) shall be a noncoindexed integer scalar with a decimal range of at least four. It is an INTENT (OUT) argument. It is assigned a value as specified in 16.5. If an error condition occurs and STAT is not present, error termination is initiated.

4 Example. If the value of I on image 3 is equal to 13 at the beginning of the atomic operation performed by CALL ATOMIC_CAS (I [3], OLD, 0, 1), the value of I on image 3 will be unchanged, and OLD will become defined with the value 13. If the value of I on image 3 is equal to 0 at the beginning of the atomic operation performed by CALL ATOMIC_CAS (I [3], OLD, 0, 1), I on image 3 will become defined with the value 1, and OLD will become defined with the value 0.

16.9.23 ATOMIC_DEFINE (ATOM, VALUE [, STAT])

1 Description. Define a variable atomically.

2 Class. Atomic subroutine.

3 Arguments.

ATOM shall be a scalar coarray or coindexed object. It shall be of type integer with kind ATOMIC_INT_KIND from the intrinsic module ISO_FORTRAN_ENV, or of type logical with kind ATOMIC_LOGICAL_KIND from the intrinsic module ISO_FORTRAN_ENV. It is an INTENT (OUT) argument. On successful execution, it becomes defined with the value of VALUE. If an error condition occurs, it becomes undefined.

VALUE shall be scalar and of the same type as ATOM. It is an INTENT (IN) argument.

STAT (optional) shall be a noncoindexed integer scalar with a decimal range of at least four. It is an INTENT (OUT) argument. It is assigned a value as specified in 16.5. If an error condition occurs and STAT is not present, error termination is initiated.

4 Example. CALL ATOMIC_DEFINE (I [3], 4) causes I on image 3 to become defined with the value 4.

16.9.24 ATOMIC_FETCH_ADD (ATOM, VALUE, OLD [, STAT])

1 Description. Atomic fetch and add.

2 Class. Atomic subroutine.

3 Arguments.

ATOM shall be a scalar coarray or coindexed object. It shall be of type integer with kind ATOMIC_INT_KIND from the intrinsic module ISO_FORTRAN_ENV, or of type logical with kind ATOMIC_LOGICAL_KIND from the intrinsic module ISO_FORTRAN_ENV. It is an INTENT (INOUT) argument. On successful execution, it becomes defined with the value of VALUE. If an error condition occurs, it becomes undefined.

VALUE shall be scalar and of the same type as ATOM. It is an INTENT (IN) argument.

STAT (optional) shall be a noncoindexed integer scalar with a decimal range of at least four. It is an INTENT (OUT) argument. It is assigned a value as specified in 16.5. If an error condition occurs and STAT is not present, error termination is initiated.

4 Example. CALL ATOMIC_FETCH_ADD (I [3], 4) causes I on image 3 to become defined with the value 4.
KIND from the intrinsic module ISO_FORTRAN_ENV. It is an INTENT (INOUT) argument. If an error condition occurs, ATOM becomes undefined; otherwise, it becomes defined with the value of ATOM + VALUE.

VALUE shall be an integer scalar. It is an INTENT (IN) argument. The values of VALUE and ATOM + VALUE shall be representable in ATOMIC_INT_KIND.

OLD shall be scalar and of the same type and kind as ATOM. It is an INTENT (OUT) argument. If an error condition occurs, it becomes undefined; otherwise, it becomes defined with the value that ATOM had at the start of the atomic operation.

STAT (optional) shall be a noncoindexed integer scalar with a decimal range of at least four. It is an INTENT (OUT) argument. It is assigned a value as specified in 16.5. If an error condition occurs and STAT is not present, error termination is initiated.

Example. CALL ATOMIC_FETCH_ADD (I [3], 7, J) will cause I on image 3 to become defined with the value 12, and J to become defined with the value 5, if the value of I [3] is 5 when the atomic operation is executed.

16.9.25 ATOMIC_FETCH_AND (ATOM, VALUE, OLD [, STAT])

Description. Atomic fetch and bitwise AND.

Class. Atomic subroutine.

Arguments.

ATOM shall be a scalar coarray or coindexed object. It shall be of type integer with kind ATOMIC_INT_KIND from the intrinsic module ISO_FORTRAN_ENV. It is an INTENT (INOUT) argument. If an error condition occurs, ATOM becomes undefined; otherwise, it becomes defined with the value of IAND (ATOM, INT (VALUE, ATOMIC_INT_KIND)).

VALUE shall be an integer scalar. It is an INTENT (IN) argument. The value of VALUE shall be representable in ATOMIC_INT_KIND.

OLD shall be scalar and of the same type and kind as ATOM. It is an INTENT (OUT) argument. If an error condition occurs, it becomes undefined; otherwise, it becomes defined with the value that ATOM had at the start of the atomic operation.

STAT (optional) shall be a noncoindexed integer scalar with a decimal range of at least four. It is an INTENT (OUT) argument. It is assigned a value as specified in 16.5. If an error condition occurs and STAT is not present, error termination is initiated.

Example. CALL ATOMIC_FETCH_AND (I [3], 6, J) will cause I on image 3 to become defined with the value 4, and J to become defined with the value 5, if the value of I [3] is 5 when the atomic operation is executed.

16.9.26 ATOMIC_FETCH_OR (ATOM, VALUE, OLD [, STAT])

Description. Atomic fetch and bitwise OR.

Class. Atomic subroutine.

Arguments.

ATOM shall be a scalar coarray or coindexed object. It shall be of type integer with kind ATOMIC_INT_KIND from the intrinsic module ISO_FORTRAN_ENV. It is an INTENT (INOUT) argument. If an error condition occurs, ATOM becomes undefined; otherwise, it becomes defined with the value of IOR (ATOM, INT (VALUE, ATOMIC_INT_KIND)).

VALUE shall be an integer scalar. It is an INTENT (IN) argument. The value of VALUE shall be representable in ATOMIC_INT_KIND.

OLD shall be scalar and of the same type and kind as ATOM. It is an INTENT (OUT) argument. If an error condition occurs, it becomes undefined; otherwise, it becomes defined with the value that ATOM had at the start of the atomic operation.
STAT (optional) shall be a noncoindexed integer scalar with a decimal range of at least four. It is an INTENT (OUT) argument. It is assigned a value as specified in 16.5. If an error condition occurs and STAT is not present, error termination is initiated.

Example. CALL ATOMIC_FETCH_OR (I [3], 1, J) will cause I on image 3 to become defined with the value 3, and J to become defined with the value 2, if the value of I [3] is 2 when the atomic operation is executed.

16.9.27 ATOMIC_FETCH_XOR (ATOM, VALUE, OLD [, STAT])

Description. Atomic fetch and bitwise exclusive OR.

Class. Atomic subroutine.

Arguments.

ATOM shall be a scalar coarray or coindexed object. It shall be of type integer with kind ATOMIC_INT_KIND from the intrinsic module ISO_FORTRAN_ENV. It is an INTENT (INOUT) argument. If an error condition occurs, ATOM becomes undefined; otherwise, it becomes defined with the value of IEOR (ATOM, INT (VALUE, ATOMIC_INT_KIND)).

VALUE shall be an integer scalar. It is an INTENT (IN) argument. The value of VALUE shall be representable in ATOMIC_INT_KIND.

OLD shall be scalar and of the same type and kind as ATOM. It is an INTENT (OUT) argument. If an error condition occurs, it becomes undefined; otherwise, it becomes defined with the value that ATOM had at the start of the atomic operation.

STAT (optional) shall be a noncoindexed integer scalar with a decimal range of at least four. It is an INTENT (OUT) argument. It is assigned a value as specified in 16.5. If an error condition occurs and STAT is not present, error termination is initiated.

Example. CALL ATOMIC_FETCH_XOR (I [3], 1, J) will cause I on image 3 to become defined with the value 2, and J to become defined with the value 3, if the value of I [3] is 3 when the atomic operation is executed.

16.9.28 ATOMIC_OR (ATOM, VALUE [, STAT])

Description. Atomic bitwise OR.

Class. Atomic subroutine.

Arguments.

ATOM shall be a scalar coarray or coindexed object. It shall be of type integer with kind ATOMIC_INT_KIND from the intrinsic module ISO_FORTRAN_ENV. It is an INTENT (INOUT) argument. If an error condition occurs, ATOM becomes undefined; otherwise, it becomes defined with the value of IOR (ATOM, INT (VALUE, ATOMIC_INT_KIND)).

VALUE shall be an integer scalar. It is an INTENT (IN) argument. The value of VALUE shall be representable in ATOMIC_INT_KIND.

STAT (optional) shall be a noncoindexed integer scalar with a decimal range of at least four. It is an INTENT (OUT) argument. It is assigned a value as specified in 16.5. If an error condition occurs and STAT is not present, error termination is initiated.

Example. CALL ATOMIC_OR (I [3], 1) will cause I on image 3 to become defined with the value 3 if the value of I [3] is 2 when the atomic operation is executed.

16.9.29 ATOMIC_REF (VALUE, ATOM [, STAT])

Description. Reference a variable atomically.

Class. Atomic subroutine.
3 Arguments.

VALUE shall be scalar and of the same type as ATOM. It is an INTENT (OUT) argument. On successful execution, it becomes defined with the value of ATOM. If an error condition occurs, it becomes undefined.

ATOM shall be a scalar coarray or coindexed object. It shall be of type integer with kind ATOMIC_INT_KIND from the intrinsic module ISO_FORTRAN_ENV, or of type logical with kind ATOMIC_LOGICAL_KIND from the intrinsic module ISO_FORTRAN_ENV. It is an INTENT (IN) argument.

STAT (optional) shall be a noncoindexed integer scalar with a decimal range of at least four. It is an INTENT (OUT) argument. It is assigned a value as specified in 16.5. If an error condition occurs and STAT is not present, error termination is initiated.

Example. CALL ATOMIC_REF (VAL, I [3]) causes VAL to become defined with the value of I on image 3.

16.9.30 ATOMIC_XOR (ATOM, VALUE [, STAT])

3 Arguments.

ATOM shall be a scalar coarray or coindexed object. It shall be of type integer with kind ATOMIC_INT_KIND from the intrinsic module ISO_FORTRAN_ENV. It is an INTENT (INOUT) argument. If an error condition occurs, ATOM becomes undefined; otherwise, it becomes defined with the value of IEOR (ATOM, INT (VALUE, ATOMIC_INT_KIND)).

VALUE shall be an integer scalar. It is an INTENT (IN) argument. The value of VALUE shall be representable in ATOMIC_INT_KIND.

STAT (optional) shall be a noncoindexed integer scalar with a decimal range of at least four. It is an INTENT (OUT) argument. It is assigned a value as specified in 16.5. If an error condition occurs and STAT is not present, error termination is initiated.

Example. CALL ATOMIC_XOR (I [3], 1) will cause I on image 3 to become defined with the value 2 if the value of I [3] is 3 when the atomic operation is executed.

16.9.31 BESSEL_J0 (X)

3 Argument. X shall be of type real.

4 Result Characteristics. Same as X.

5 Result Value. The result has a value equal to a processor-dependent approximation to the Bessel function of the first kind and order zero of X.

Example. BESSEL_J0 (1.0) has the value 0.765 (approximately).

16.9.32 BESSEL_J1 (X)

3 Argument. X shall be of type real.

4 Result Characteristics. Same as X.
Result Value. The result has a value equal to a processor-dependent approximation to the Bessel function of the first kind and order one of X.

Example. BESSEL_J1 (1.0) has the value 0.440 (approximately).

16.9.33 BESSEL_JN (N, X) or BESSEL_JN (N1, N2, X)

Description. Bessel functions of the 1st kind.

Class.

Case (i): BESSEL_JN (N,X) is an elemental function.

Case (ii): BESSEL_JN (N1,N2,X) is a transformational function.

Arguments.

N shall be of type integer and nonnegative.
N1 shall be an integer scalar with a nonnegative value.
N2 shall be an integer scalar with a nonnegative value.
X shall be of type real; if the function is transformational, X shall be scalar.

Result Characteristics. Same type and kind as X. The result of BESSEL_JN (N1, N2, X) is a rank-one array with extent MAX (N2−N1+1, 0).

Result Value.

Case (i): The result value of BESSEL_JN (N, X) is a processor-dependent approximation to the Bessel function of the first kind and order N of X.

Case (ii): Element i of the result value of BESSEL_JN (N1, N2, X) is a processor-dependent approximation to the Bessel function of the first kind and order N1+i−1 of X.

Example. BESSEL_JN (2, 1.0) has the value 0.115 (approximately).

16.9.34 BESSEL_Y0 (X)

Description. Bessel function of the 2nd kind, order 0.

Class. Elemental function.

Argument. X shall be of type real. Its value shall be greater than zero.

Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to the Bessel function of the second kind and order zero of X.

Example. BESSEL_Y0 (1.0) has the value 0.088 (approximately).

16.9.35 BESSEL_Y1 (X)

Description. Bessel function of the 2nd kind, order 1.

Class. Elemental function.

Argument. X shall be of type real. Its value shall be greater than zero.

Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to the Bessel function of the second kind and order one of X.
Example. BESSEL_Y1 (1.0) has the value $-0.781$ (approximately).

16.9.36 BESSEL_YN (N, X) or BESSEL_YN (N1, N2, X)

Description. Bessel functions of the 2nd kind.

Class.

Case (i): BESSEL_YN (N, X) is an elemental function.

Case (ii): BESSEL_YN (N1, N2, X) is a transformational function.

Arguments.

N shall be of type integer and nonnegative.

N1 shall be an integer scalar with a nonnegative value.

N2 shall be an integer scalar with a nonnegative value.

X shall be of type real; if the function is transformational, X shall be scalar. Its value shall be greater than zero.

Result Characteristics. Same type and kind as X. The result of BESSEL_YN (N1, N2, X) is a rank-one array with extent MAX (N2−N1+1, 0).

Result Value.

Case (i): The result value of BESSEL_YN (N, X) is a processor-dependent approximation to the Bessel function of the second kind and order N of X.

Case (ii): Element $i$ of the result value of BESSEL_YN (N1, N2, X) is a processor-dependent approximation to the Bessel function of the second kind and order N1+$i$ − 1 of X.

Example. BESSEL_YN (2, 1.0) has the value $-1.651$ (approximately).

16.9.37 BGE (I, J)

Description. Bitwise greater than or equal to.

Class. Elemental function.

Arguments.

I shall be of type integer or a boz-literal-constant.

J shall be of type integer or a boz-literal-constant.

Result Characteristics. Default logical.

Result Value. The result is true if the sequence of bits represented by I is greater than or equal to the sequence of bits represented by J, according to the method of bit sequence comparison in 16.3.2; otherwise the result is false.

Example. If BIT_SIZE (J) has the value 8, BGE (Z’FF’, J) has the value true for any value of J. BGE (0, $-1$) has the value false.

16.9.38 BGT (I, J)

Description. Bitwise greater than.

Class. Elemental function.

Arguments.
I shall be of type integer or a \textit{boz-literal-constant}.

J shall be of type integer or a \textit{boz-literal-constant}.

4 Result Characteristics. Default logical.

5 Result Value. The result is true if the sequence of bits represented by I is greater than the sequence of bits represented by J, according to the method of bit sequence comparison in 16.3.2; otherwise the result is false.

6 The interpretation of a \textit{boz-literal-constant} as a sequence of bits is described in 7.7. The interpretation of an integer value as a sequence of bits is described in 16.3.

7 Example. BGT (Z'FF', Z'FC') has the value true. BGT (0, −1) has the value false.

16.9.39 BIT\_SIZE (I)

1 Description. Number of bits in integer model 16.3.

2 Class. Inquiry function.

3 Argument. I shall be of type integer. It may be a scalar or an array.

4 Result Characteristics. Scalar integer with the same kind type parameter as I.

5 Result Value. The result has the value of the number of bits $z$ of the model integer defined for bit manipulation contexts in 16.3.

6 Example. BIT\_SIZE (1) has the value 32 if $z$ of the model is 32.

16.9.40 BLE (I, J)

1 Description. Bitwise less than or equal to.

2 Class. Elemental function.

3 Arguments.

4 Result Characteristics. Default logical.

5 Result Value. The result is true if the sequence of bits represented by I is less than or equal to the sequence of bits represented by J, according to the method of bit sequence comparison in 16.3.2; otherwise the result is false.

6 The interpretation of a \textit{boz-literal-constant} as a sequence of bits is described in 7.7. The interpretation of an integer value as a sequence of bits is described in 16.3.

7 Example. BLE (0, J) has the value true for any value of J. BLE (−1, 0) has the value false.

16.9.41 BLT (I, J)

1 Description. Bitwise less than.

2 Class. Elemental function.

3 Arguments.

4 Result Characteristics. Default logical.
5 **Result Value.** The result is true if the sequence of bits represented by I is less than the sequence of bits represented by J, according to the method of bit sequence comparison in 16.3.2; otherwise, the result is false.

6 The interpretation of a *boz-literal-constant* as a sequence of bits is described in 7.7. The interpretation of an integer value as a sequence of bits is described in 16.3.

7 **Example.** BLT (0, −1) has the value true. BLT (Z′FF′, Z′FC′) has the value false.

### 16.9.42 BTEST (I, POS)

1 **Description.** Test single bit in an integer.

2 **Class.** Elemental function.

3 **Arguments.**

4 **I** shall be of type integer.

5 **POS** shall be of type integer. It shall be nonnegative and be less than BIT_SIZE (I).

4 **Result Characteristics.** Default logical.

5 **Result Value.** The result has the value true if bit POS of I has the value 1 and has the value false if bit POS of I has the value 0. The model for the interpretation of an integer value as a sequence of bits is in 16.3.

6 **Examples.** BTEST (8, 3) has the value true. If A has the value $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, the value of BTEST (A, 2) is $\begin{bmatrix} \text{false} & \text{false} \\ \text{false} & \text{true} \end{bmatrix}$ and the value of BTEST (2, A) is $\begin{bmatrix} \text{true} & \text{false} \\ \text{false} & \text{false} \end{bmatrix}$.

### 16.9.43 CEILING (A [, KIND])

1 **Description.** Least integer greater than or equal to A.

2 **Class.** Elemental function.

3 **Arguments.**

4 **A** shall be of type real.

5 **KIND** (optional) shall be a scalar integer constant expression.

4 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default integer type.

5 **Result Value.** The result has a value equal to the least integer greater than or equal to A.

6 **Examples.** CEILING (3.7) has the value 4. CEILING (−3.7) has the value −3.

### 16.9.44 CHAR (I [, KIND])

1 **Description.** Character from code value.

2 **Class.** Elemental function.

3 **Arguments.**

4 **I** shall be of type integer with a value in the range $0 \leq I \leq n − 1$, where $n$ is the number of characters in the collating sequence associated with the specified kind type parameter.

5 **KIND** (optional) shall be a scalar integer constant expression.

4 **Result Characteristics.** Character of length one. If KIND is present, the kind type parameter is that specified...
by the value of KIND; otherwise, the kind type parameter is that of default character.

5 **Result Value.** The result is the character in position \( I \) of the collating sequence associated with the specified kind type parameter. \( \text{ICHAR} \left( \text{CHAR} \left( I, \text{KIND} \left( C \right) \right) \right) \) shall have the value \( I \) for \( 0 \leq I \leq n - 1 \) and \( \text{CHAR} \left( \text{ICHAR} \left( C \right), \text{KIND} \left( C \right) \right) \) shall have the value \( C \) for any character \( C \) capable of representation in the processor.

6 **Example.** \( \text{CHAR} \left( 88 \right) \) has the value ‘\( \text{X} \)’ on a processor using the ASCII collating sequence for default characters.

16.9.45 **CMPLX \( \left( X \right[, \text{KIND} \right) \text{ or } \text{CMPLX} \left( X \right[, Y \text{, KIND}) \) **

1 **Description.** Conversion to complex type.

2 **Class.** Elemental function.

3 **Arguments for CMPLX(\( X \left[, \text{KIND} \right) \).**
   4 \( X \) shall be of type complex.
   5 KIND (optional) shall be a scalar integer constant expression.

4 **Arguments for CMPLX(\( X \left[, Y \text{, KIND}) \).**
   6 \( X \) shall be of type integer or real, or a `boz-literal-constant`.
   7 \( Y \) (optional) shall be of type integer or real, or a `boz-literal-constant`.
   8 KIND (optional) shall be a scalar integer constant expression.

5 **Result Characteristics.** The result is of type complex. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default real kind.

6 **Result Value.** If \( Y \) is absent and \( X \) is not complex, it is as if \( Y \) were present with the value zero. If KIND is absent, it is as if KIND were present with the value \( \text{KIND} \left( 0.0 \right) \). If \( X \) is complex, the result is the same as that of CMPLX (\( \text{REAL} \left( X \right), \text{AIMAG} \left( X \right), \text{KIND} \right) \). The result of CMPLX (\( X \), \( Y \), \( \text{KIND} \right) \) has the complex value whose real part is \( \text{REAL} \left( X, \text{KIND} \right) \) and whose imaginary part is \( \text{REAL} \left( Y, \text{KIND} \right) \).

7 **Example.** CMPLX (\( -3 \)) has the value \( (-3.0, 0.0) \).

16.9.46 **CO_BROADCAST(\( A \), SOURCE_IMAGE \left[, \text{STAT}, \text{ERRMSG} \right) \)**

1 **Description.** Broadcast value to images.

2 **Class.** Collective subroutine.

3 **Arguments.**
   4 \( A \) shall have the same shape, dynamic type, and type parameter values, in corresponding references.
   5 It shall not be a coindexed object. It is an \text{INTENT} (\text{INOUT}) argument. If no error condition occurs, \( A \) becomes defined, as if by intrinsic assignment, on all images in the current team with the value of \( A \) on image SOURCE_IMAGE.
   6 SOURCE_IMAGE shall be an integer scalar. It is an \text{INTENT} (\text{IN}) argument. Its value shall be that of an image index of an image in the current team. The value shall be the same in all corresponding references.
   7 STAT (optional) shall be a noncoindexed integer scalar with a decimal exponent range of at least four. It is an \text{INTENT} (\text{OUT}) argument.
   8 ERRMSG (optional) shall be a noncoindexed default character scalar. It is an \text{INTENT} (\text{INOUT}) argument.

4 The semantics of STAT and ERRMSG are described in 16.6.

5 **Example.** If \( A \) is the array \([1, 5, 3]\) on image one, after execution of
CALL CO_BROADCAST (A, 1)

the value of A on all images of the current team is [1, 5, 3].

16.9.47 CO_MAX(A [, RESULT_IMAGE, STAT, ERRMSG])

1 Description. Compute maximum value across images.

2 Class. Collective subroutine.

3 Arguments.

A shall be of type integer, real, or character. It shall have the same shape, type, and type parameter
values, in corresponding references. It shall not be a coindexed object. It is an INTENT (INOUT)
argument. If it is scalar, the computed value is equal to the maximum value of A in all corresponding
references. If it is an array each element of the computed value is equal to the maximum value of
all corresponding elements of A in all corresponding references.

The computed value is assigned to A if no error condition occurs, and either RESULT_IMAGE is
absent, or the executing image is the one identified by RESULT_IMAGE. Otherwise, A becomes
undefined.

RESULT_IMAGE (optional) shall be an integer scalar. It is an INTENT (IN) argument. Its presence, and value
if present, shall be the same in all corresponding references. If it is present, its value shall be that
of an image index in the current team.

STAT (optional) shall be a noncoindexed integer scalar with a decimal exponent range of at least four. It is an
INTENT (OUT) argument.

ERRMSG (optional) shall be a noncoindexed default character scalar. It is an INTENT (INOUT) argument.

4 The semantics of STAT and ERRMSG are described in 16.6.

5 Example. If the number of images in the current team is two and A is the array [1, 5, 3] on one image and [4,
1, 6] on the other image, the value of A after executing the statement CALL CO_MAX (A) is [4, 5, 6] on both
images.

16.9.48 CO_MIN(A [, RESULT_IMAGE, STAT, ERRMSG])

1 Description. Compute minimum value across images.

2 Class. Collective subroutine.

3 Arguments.

A shall be of type integer, real, or character. It shall have the same shape, type, and type parameter
values, in corresponding references. It shall not be a coindexed object. It is an INTENT (INOUT)
argument. If it is scalar, the computed value is equal to the minimum value of A in all corresponding
references. If it is an array each element of the computed value is equal to the minimum value of
all corresponding elements of A in all corresponding references.

The computed value is assigned to A if no error condition occurs, and either RESULT_IMAGE is
absent, or the executing image is the one identified by RESULT_IMAGE. Otherwise, A becomes
undefined.

RESULT_IMAGE (optional) shall be an integer scalar. It is an INTENT (IN) argument. Its presence, and value
if present, shall be the same in all corresponding references. If it is present, its value shall be that
of an image index in the current team.

STAT (optional) shall be a noncoindexed integer scalar with a decimal exponent range of at least four. It is an
INTENT (OUT) argument.

ERRMSG (optional) shall be a noncoindexed default character scalar. It is an INTENT (INOUT) argument.

4 The semantics of STAT and ERRMSG are described in 16.6.
Example. If the number of images in the current team is two and A is the array [1, 5, 3] on one image and [4, 1, 6] on the other image, the value of A after executing the statement CALL CO_MIN (A) is [1, 1, 3] on both images.

16.9.49 CO_REDUCE(A, OPERATOR [, RESULT_IMAGE, STAT, ERRMSG])

Description. Generalized reduction across images.

Class. Collective subroutine.

Arguments.

A shall not be polymorphic. It shall have the same shape, type, and type parameter values, in corresponding references. It shall not be a coindexed object. It is an INTENT (INOUT) argument. If A is scalar, the computed value is the result of the reduction operation of applying OPERATOR to the values of A in all corresponding references. If A is an array, each element of the computed value is equal to the result of the reduction operation of applying OPERATOR to corresponding elements of A in all corresponding references.

The computed value is assigned to A if no error condition occurs, and either RESULT_IMAGE is absent, or the executing image is the one identified by RESULT_IMAGE. Otherwise, A becomes undefined.

OPERATOR shall be a pure function with two scalar, nonallocatable, nonpointer, nonpolymorphic, nonoptional arguments of the same type and type parameters as A. Its result shall have the same type and type parameters as A. If one argument has the ASYNCHRONOUS, TARGET, or VALUE attribute, the other shall have that attribute. OPERATOR shall implement a mathematically associative operation. OPERATOR shall be the same function on all images in corresponding references.

The computed value of a reduction operation over a set of values is the result of an iterative process. Each iteration involves the evaluation of OPERATOR (x, y) for x and y in the set, the removal of x and y from the set, and the addition of the value of OPERATOR (x, y) to the set. The process terminates when the set has only one element; this is the computed value.

RESULT_IMAGE (optional) shall be an integer scalar. It is an INTENT (IN) argument. Its presence, and value if present, shall be the same in all corresponding references. If it is present, its value shall be that of an image index in the current team.

STAT (optional) shall be a noncoindexed integer scalar with a decimal exponent range of at least four. It is an INTENT (OUT) argument.

ERRMSG (optional) shall be a noncoindexed default character scalar. It is an INTENT (INOUT) argument.

The semantics of STAT and ERRMSG are described in 16.6.

Example. The subroutine below demonstrates how to use CO_REDUCE to create a collective counterpart to the intrinsic function ALL:

```
SUBROUTINE co_all(boolean)
LOGICAL, INTENT(INOUT) :: boolean
CALL CO_REDUCE(boolean,both)
END SUBROUTINE co_all
```

ISO/IEC JTC 1/SC 22/WG5/N2118 371
NOTE 16.11
If the OPERATOR function is not mathematically commutative, the result of calling CO_REDUCE can depend on the order of evaluations.

16.9.50  CO_SUM(A [, RESULT_IMAGE, STAT, ERRMSG])

Description. Compute sum across images.

Class. Collective subroutine.

Arguments.

A shall be of numeric type. It shall have the same shape, type, and type parameter values, in corresponding references. It shall not be a coindexed object. It is an INTENT (INOUT) argument. If it is scalar, the computed value is equal to a processor-dependent approximation to the sum of the values of A in corresponding references. If it is an array, each element of the computed value is equal to a processor-dependent approximation to the sum of all corresponding elements of A in corresponding references.

The computed value is assigned to A if no error condition occurs, and either RESULT_IMAGE is absent, or the executing image is the one identified by RESULT_IMAGE. Otherwise, A becomes undefined.

RESULT_IMAGE (optional) shall be an integer scalar. It is an INTENT (IN) argument. Its presence, and value if present, shall be the same in all corresponding references. If it is present, its value shall be that of an image index in the current team.

STAT (optional) shall be a noncoindexed integer scalar with a decimal exponent range of at least four. It is an INTENT (OUT) argument.

ERRMSG (optional) shall be a noncoindexed default character scalar. It is an INTENT (INOUT) argument.

Example. If the number of images in the current team is two and A is the array [1, 5, 3] on one image and [4, 1, 6] on the other image, the value of A after executing the statement CALL CO_SUM(A) is [5, 6, 9] on both images.

16.9.51  COMMAND_ARGUMENT_COUNT ( )

Description. Number of command arguments.

Class. Transformational function.

Argument. None.

Result Characteristics. Default integer scalar.

Result Value. The result value is equal to the number of command arguments available. If there are no command arguments available or if the processor does not support command arguments, then the result has the value zero. If the processor has a concept of a command name, the command name does not count as one of the command arguments.

Example. See 16.9.83.

16.9.52  CONJG (Z)

Description. Conjugate of a complex number.

Class. Elemental function.
Argument. Z shall be of type complex.

Result Characteristics. Same as Z.

Result Value. If Z has the value \((x, y)\), the result has the value \((x, -y)\).

Example. \(\text{CONJG}((2.0, 3.0))\) has the value \((2.0, -3.0)\).

16.9.53 \(\text{COS} (X)\)

Description. Cosine function.

Class. Elemental function.

Argument. X shall be of type real or complex.

Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to \(\cos(X)\). If X is of type real, it is regarded as a value in radians. If X is of type complex, its real part is regarded as a value in radians.

Example. \(\text{COS} (1.0)\) has the value 0.54030231 (approximately).

16.9.54 \(\text{COSH} (X)\)

Description. Hyperbolic cosine function.

Class. Elemental function.

Argument. X shall be of type real or complex.

Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to \(\cosh(X)\). If X is of type complex its imaginary part is regarded as a value in radians.

Example. \(\text{COSH} (1.0)\) has the value 1.5430806 (approximately).

16.9.55 \(\text{COSHAPE} (\text{COARRAY [, KIND]}))\)

Description. Sizes of codimensions of a coarray.

Class. Inquiry function.

Arguments.

\text{COARRAY} shall be a coarray of any type. It shall not be an unallocated allocatable coarray.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type. The result is an array of rank one whose size is equal to the corank of \text{COARRAY}.

Result Value. The result has a value whose \(i^{th}\) element is equal to the size of the \(i^{th}\) codimension of \text{COARRAY}, as given by \(\text{UCOBOUND(\text{COARRAY},i)} - \text{LCOBOUND(\text{COARRAY},i)} + 1\).

Example.

The following code allocates the coarray D with the same size in each codimension as that of the coarray C, with the lower cobound 1.

\begin{verbatim}
REAL, ALLOCATABLE :: C[:,:], D[:,:]
INTEGER, ALLOCATABLE :: COSHAPE_C(:)
\end{verbatim}
16.9.56 COUNT (MASK [, DIM, KIND])

Description. Logical array reduced by counting true values.

Class. Transformational function.

Arguments.

MASK shall be a logical array.

DIM (optional) shall be an integer scalar with a value in the range $1 \leq \text{DIM} \leq n$, where $n$ is the rank of MASK. The corresponding actual argument shall not be an optional dummy argument, a disassociated pointer, or an unallocated allocatable.

KIND (optional) shall be a scalar integer constant expression.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type. The result is scalar if DIM is absent or $n = 1$; otherwise, the result has rank $n - 1$ and shape $[d_1, d_2, \ldots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \ldots, d_n]$ where $[d_1, d_2, \ldots, d_n]$ is the shape of MASK.

Result Value.

Case (i): If DIM is absent or MASK has rank one, the result has a value equal to the number of true elements of MASK or has the value zero if MASK has size zero.

Case (ii): If DIM is present and MASK has rank $n > 1$, the value of element $(s_1, s_2, \ldots, s_{\text{DIM}-1}, \text{DIM}+1, \ldots, s_n)$ of the result is equal to the number of true elements of MASK $(s_1, s_2, \ldots, s_{\text{DIM}-1}, :,$ $s_{\text{DIM}+1}, \ldots, s_n)$.

Examples.

Case (i): The value of COUNT ([.TRUE., .FALSE., .TRUE.]) is 2.

Case (ii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ and C is the array $\begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}$, COUNT (B $\neq$ C, DIM = 1) is [2, 0, 1] and COUNT (B $\neq$ C, DIM = 2) is [1, 2].

16.9.57 CPU_TIME (TIME)

Description. Processor time used.

Class. Subroutine.

Argument. TIME shall be a real scalar. It is an INTENT (OUT) argument. If the processor cannot provide a meaningful value for the time, it is assigned a processor-dependent negative value; otherwise, it is assigned a processor-dependent approximation to the processor time in seconds. Whether the value assigned is an approximation to the amount of time used by the invoking image, or the amount of time used by the whole program, is processor dependent.

Example.

REAL T1, T2

CALL CPU_TIME(T1)

CALL CPU_TIME(T2)

WRITE (*,*) 'Time taken by code was ', T2-T1, ' seconds'
writes the processor time taken by a piece of code.

NOTE 16.12

A processor for which a single result is inadequate (for example, a parallel processor) might choose to provide an additional version for which time is an array.

The exact definition of time is left imprecise because of the variability in what different processors are able to provide. The primary purpose is to compare different algorithms on the same processor or discover which parts of a calculation are the most expensive.

The start time is left imprecise because the purpose is to time sections of code, as in the example.

Most computer systems have multiple concepts of time. One common concept is that of time expended by the processor for a given program. This might or might not include system overhead, and has no obvious connection to elapsed “wall clock” time.

16.9.58 CSHIFT (ARRAY, SHIFT [, DIM])

1 Description. Circular shift of an array.

2 Class. Transformational function.

3 Arguments.

ARRAY may be of any type. It shall be an array.

SHIFT shall be of type integer and shall be scalar if ARRAY has rank one; otherwise, it shall be scalar or of rank n - 1 and of shape [d1, d2, ..., dDIM−1, dDIM+1, ..., dn] where [d1, d2, ..., dn] is the shape of ARRAY.

DIM (optional) shall be an integer scalar with a value in the range 1 ≤ DIM ≤ n, where n is the rank of ARRAY.

If DIM is absent, it is as if it were present with the value 1.

4 Result Characteristics. The result is of the type and type parameters of ARRAY, and has the shape of ARRAY.

5 Result Value.

Case (i): If ARRAY has rank one, element i of the result is ARRAY (1 + MODULO (i + SHIFT − 1, SIZE (ARRAY))).

Case (ii): If ARRAY has rank greater than one, section (s1, s2, ..., sDIM−1, ;, sDIM+1, ..., sn) of the result has a value equal to CSHIFT (ARRAY (s1, s2, ..., sDIM−1, ;, sDIM+1, ..., sn), sh, 1), where sh is SHIFT or SHIFT (s1, s2, ..., sDIM−1, sDIM+1, ..., sn).

6 Examples.

Case (i): If V is the array [1, 2, 3, 4, 5, 6], the effect of shifting V circularly to the left by two positions is achieved by CSHIFT (V, SHIFT = 2) which has the value [3, 4, 5, 6, 1, 2]; CSHIFT (V, SHIFT = −2) achieves a circular shift to the right by two positions and has the value [5, 6, 1, 2, 3, 4].

Case (ii): The rows of an array of rank two may all be shifted by the same amount or by different amounts.

If M is the array

\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{bmatrix}
\]

, the value of

CSHIFT (M, SHIFT = −1, DIM = 2) is

\[
\begin{bmatrix}
3 & 1 & 2 \\
6 & 4 & 5 \\
9 & 7 & 8
\end{bmatrix}
\]

, and the value of

CSHIFT (M, SHIFT = [−1, 1, 0], DIM = 2) is

\[
\begin{bmatrix}
3 & 1 & 2 \\
5 & 6 & 4 \\
7 & 8 & 9
\end{bmatrix}
\]
16.9.59 DATE_AND_TIME ([DATE, TIME, ZONE, VALUES])

1 Description. Date and time.

2 Class. Subroutine.

3 Arguments.

4 DATE (optional) shall be a default character scalar. It is an INTENT (OUT) argument. It is assigned a value of the form YYYYMMDD, where YYYY is the year in the Gregorian calendar, MM is the month within the year, and DD is the day within the month. The characters of this value shall all be decimal digits. If there is no date available, DATE is assigned all blanks.

5 TIME (optional) shall be a default character scalar. It is an INTENT (OUT) argument. It is assigned 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. Except for the decimal point, the characters of this value shall all be decimal digits. If there is no clock available, TIME is assigned all blanks.

6 ZONE (optional) shall be a default character scalar. It is an INTENT (OUT) argument. It is assigned a value of the form +hhmm or -hhmm, where hh and mm are the time difference with respect to Coordinated Universal Time (UTC) in hours and minutes, respectively. The characters of this value following the sign character shall all be decimal digits. If this information is not available, ZONE is assigned all blanks.

7 VALUES (optional) shall be a rank-one array of type integer with a decimal exponent range of at least four. It is an INTENT (OUT) argument. Its size shall be at least 8. The values assigned to VALUES are as follows:

VALUES (1) the year, including the century (for example, 2008), or −HUGE (VALUES) if there is no date available;

VALUES (2) the month of the year, or −HUGE (VALUES) if there is no date available;

VALUES (3) the day of the month, or −HUGE (VALUES) if there is no date available;

VALUES (4) the time difference with respect to Coordinated Universal Time (UTC) in minutes, or −HUGE (VALUES) if this information is not available;

VALUES (5) the hour of the day, in the range of 0 to 23, or −HUGE (VALUES) if there is no clock;

VALUES (6) the minutes of the hour, in the range 0 to 59, or −HUGE (VALUES) if there is no clock;

VALUES (7) the seconds of the minute, in the range 0 to 60, or −HUGE (VALUES) if there is no clock;

VALUES (8) the milliseconds of the second, in the range 0 to 999, or −HUGE (VALUES) if there is no clock.

4 The date, clock, and time zone information might be available on some images and not others. If the date, clock, or time zone information is available on more than one image, it is processor dependent whether or not those images share the same information.

5 Example. If run in Geneva, Switzerland on April 12, 2008 at 15:27:35.5 with a system configured for the local time zone, this example would have assigned the value 20080412 to BIG_BEN (1), the value 152735.500 to BIG_BEN (2), the value +0100 to BIG_BEN (3), and the value [2008, 4, 12, 60, 15, 27, 35, 500] to DATE_TIME.

   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)

NOTE 16.13

These forms are compatible with the representations defined in ISO 8601:2004. UTC is established by the International Bureau of Weights and Measures (BIPM, i.e. Bureau International des Poids et Mesures) and the International Earth Rotation Service (IERS).
16.9.60 DBLE (A)

Description. Conversion to double precision real.

Class. Elemental function.

Argument. A shall be of type integer, real, complex, or a boz-literal-constant.

Result Characteristics. Double precision real.

Result Value. The result has the value REAL (A, KIND (0.0D0)).

Example. DBLE (−3) has the value −3.0D0.

16.9.61 DIGITS (X)

Description. Significant digits in numeric model.

Class. Inquiry function.

Argument. X shall be of type integer or real. It may be a scalar or an array.

Result Characteristics. Default integer scalar.

Result Value. The result has the value q if X is of type integer and p if X is of type real, where q and p are as defined in 16.4 for the model representing numbers of the same type and kind type parameter as X.

Example. DIGITS (X) has the value 24 for real X whose model is as in Note 16.4.

16.9.62 DIM (X, Y)

Description. Maximum of X − Y and zero.

Class. Elemental function.

Arguments.

X shall be of type integer or real.

Y shall be of the same type and kind type parameter as X.

Result Characteristics. Same as X.

Result Value. The value of the result is the maximum of X − Y and zero.

Example. DIM (−3.0, 2.0) has the value 0.0.

16.9.63 DOT_PRODUCT (VECTOR_A, VECTOR_B)

Description. Dot product of two vectors.

Class. Transformational function.

Arguments.

VECTOR_A shall be of numeric type (integer, real, or complex) or of logical type. It shall be a rank-one array.

VECTOR_B shall be of numeric type if VECTOR_A is of numeric type or of type logical if VECTOR_A is of type logical. It shall be a rank-one array. It shall be of the same size as VECTOR_A.

Result Characteristics. If the arguments are of numeric type, the type and kind type parameter of the result are those of the expression VECTOR_A * VECTOR_B determined by the types and kinds of the arguments.
according to 10.1.9.3. If the arguments are of type logical, the result is of type logical with the kind type parameter of the expression VECTOR_A .AND. VECTOR_B according to 10.1.9.3. The result is scalar.

5 Result Value.

Case (i): If VECTOR_A is of type integer or real, the result has the value \( \text{SUM} (\text{VECTOR}_A*\text{VECTOR}_B) \). If the vectors have size zero, the result has the value zero.

Case (ii): If VECTOR_A is of type complex, the result has the value \( \text{SUM} (\text{CONJG} (\text{VECTOR}_A)*\text{VECTOR}_B) \). If the vectors have size zero, the result has the value zero.

Case (iii): If VECTOR_A is of type logical, the result has the value \( \text{ANY} (\text{VECTOR}_A .AND. \text{VECTOR}_B) \). If the vectors have size zero, the result has the value false.

Example. DOT_PRODUCT ([1, 2, 3], [2, 3, 4]) has the value 20.

16.9.64 DPROD (X, Y)

1 Description. Double precision real product.

2 Class. Elemental function.

3 Arguments.

X shall be default real.

Y shall be default real.

4 Result Characteristics. Double precision real.

5 Result Value. The result has a value equal to a processor-dependent approximation to the product of X and Y. DPROD (X, Y) should have the same value as DBLE (X) * DBLE (Y).

6 Example. DPROD (-3.0, 2.0) has the value \(-6.0D0\).

16.9.65 DSHIFTL (I, J, SHIFT)

1 Description. Combined left shift.

2 Class. Elemental function.

3 Arguments.

I shall be of type integer or a \textit{boz-literal-constant}.

J shall be of type integer or a \textit{boz-literal-constant}. If both I and J are of type integer, they shall have the same kind type parameter. I and J shall not both be \textit{boz-literal-constants}.

SHIFT shall be of type integer. It shall be nonnegative and less than or equal to BIT\_SIZE (I) if I is of type integer; otherwise, it shall be less than or equal to BIT\_SIZE (J).

4 Result Characteristics. Same as I if I is of type integer; otherwise, same as J.

5 Result Value. If either I or J is a \textit{boz-literal-constant}, it is first converted as if by the intrinsic function INT to type integer with the kind type parameter of the other. The rightmost SHIFT bits of the result value are the same as the leftmost bits of J, and the remaining bits of the result value are the same as the rightmost bits of I. This is equal to IOR (SHIFTL (I, SHIFT), SHIFTR (J, BIT\_SIZE (J)−SHIFT)). The model for the interpretation of an integer value as a sequence of bits is in 16.3.

6 Examples. DSHIFTL (1, 2**30, 2) has the value 5 if default integer has 32 bits. DSHIFTL (I, I, SHIFT) has the same result value as ISHFTC (I, SHIFT).
16.9.66  **DSHIFTR (I, J, SHIFT)**

1. **Description.** Combined right shift.

2. **Class.** Elemental function.

3. **Arguments.**
   - I shall be of type integer or a *boz-literal-constant*.
   - J shall be of type integer or a *boz-literal-constant*. If both I and J are of type integer, they shall have the same kind type parameter. I and J shall not both be *boz-literal-constants*.
   - SHIFT shall be of type integer. It shall be nonnegative and less than or equal to BIT_SIZE (I) if I is of type integer; otherwise, it shall be less than or equal to BIT_SIZE (J).

4. **Result Characteristics.** Same as I if I is of type integer; otherwise, same as J.

5. **Result Value.** If either I or J is a *boz-literal-constant*, it is first converted as if by the intrinsic function INT to type integer with the kind type parameter of the other. The leftmost SHIFT bits of the result value are the same as the rightmost bits of I, and the remaining bits of the result value are the same as the leftmost bits of J. This is equal to IOR (SHIFTL (I, BIT_SIZE (I) − SHIFT), SHIFTR (J, SHIFT)). The model for the interpretation of an integer value as a sequence of bits is in 16.3.

6. **Examples.** DSHIFTR (1, 16, 3) has the value $2^{29} + 2$ if default integer has 32 bits. DSHIFTR (I, I, SHIFT) has the same result value as ISHFTC (I, −SHIFT).

16.9.67  **EOSHIFT (ARRAY, SHIFT [, BOUNDARY, DIM])**

1. **Description.** End-off shift of the elements of an array.

2. **Class.** Transformational function.

3. **Arguments.**
   - ARRAY shall be an array be of any type.
   - SHIFT shall be of type integer and shall be scalar if ARRAY has rank one; otherwise, it shall be scalar or of rank $n-1$ and of shape $[d_1, d_2, \ldots, d_{DIM-1}, d_{DIM+1}, \ldots, d_n]$ where $[d_1, d_2, \ldots, d_n]$ is the shape of ARRAY.
   - BOUNDARY (optional) shall be of the same type and type parameters as ARRAY and shall be scalar if ARRAY has rank one; otherwise, it shall be either scalar or of rank $n-1$ and of shape $[d_1, d_2, \ldots, d_{DIM-1}, d_{DIM+1}, \ldots, d_n]$. BOUNDARY is permitted to be absent only for the types in Table 16.4, and in this case it is as if it were present with the scalar value shown, converted if necessary to the kind type parameter value of ARRAY.
   - DIM (optional) shall be an integer scalar with a value in the range $1 \leq DIM \leq n$, where $n$ is the rank of ARRAY.
   - If DIM is absent, it is as if it were present with the value 1.

4. **Result Characteristics.** The result has the type, type parameters, and shape of ARRAY.

5. **Result Value.** Element $(s_1, s_2, \ldots, s_n)$ of the result has the value $\text{ARRAY} (s_1, s_2, \ldots, s_{DIM-1}, s_{DIM} + sh)$,

<table>
<thead>
<tr>
<th>Type of ARRAY</th>
<th>Value of BOUNDARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>0</td>
</tr>
<tr>
<td>Real</td>
<td>0.0</td>
</tr>
<tr>
<td>Complex</td>
<td>(0.0, 0.0)</td>
</tr>
<tr>
<td>Logical</td>
<td>.FALSE.</td>
</tr>
<tr>
<td>Character (len)</td>
<td>len blanks</td>
</tr>
</tbody>
</table>
$s_{DIM+1}, \ldots, s_n$ where $sh$ is SHIFT or SHIFT ($s_1, s_2, \ldots, s_{DIM-1}, s_{DIM+1}, \ldots, s_n$) provided the inequality 
LBOUND (ARRAY, DIM) $\leq s_{DIM} + sh \leq$ UBOUND (ARRAY, DIM) holds and is otherwise BOUNDARY or 
BOUNDARY ($s_1, s_2, \ldots, s_{DIM-1}, s_{DIM+1}, \ldots, s_n$).

Examples.

Case (i): If V is the array [1, 2, 3, 4, 5, 6], the effect of shifting V end-off to the left by 3 positions is achieved
by EOSHIFT (V, SHIFT = 3), which has the value [4, 5, 6, 0, 0, 0]; EOSHIFT (V, SHIFT = −2, 
BOUNDARY = 99) achieves an end-off shift to the right by 2 positions with the boundary value of
99 and has the value [99, 99, 1, 2, 3, 4].

Case (ii): The rows of an array of rank two may all be shifted by the same amount or by different amounts
and the boundary elements can be the same or different. If M is the array
\[
\begin{bmatrix}
A & B & C \\
D & E & F \\
G & H & I \\
\end{bmatrix}
\]
, then the
value of EOSHIFT (M, SHIFT = −1, BOUNDARY = '*'), DIM = 2) is
\[
\begin{bmatrix}
* & A & B \\
* & D & E \\
* & G & H \\
\end{bmatrix}
\]
, and the value
of EOSHIFT (M, SHIFT = [−1, 1, 0], BOUNDARY = ['*', '/', '?'], DIM = 2) is
\[
\begin{bmatrix}
* & A & B \\
E & F & / \\
G & H & I \\
\end{bmatrix}
\]

16.9.68 EPIsON (X)

1 Description. Model number that is small compared to 1.

2 Class. Inquiry function.

3 Argument. X shall be of type real. It may be a scalar or an array.

4 Result Characteristics. Scalar of the same type and kind type parameter as X.

5 Result Value. The result has the value $b^{1-p}$ where $b$ and $p$ are as defined in 16.4 for the model representing
numbers of the same type and kind type parameter as X.

6 Example. EPSILON (X) has the value $2^{-23}$ for real X whose model is as in Note 16.4.

16.9.69 ERF (X)

1 Description. Error function.

2 Class. Elemental function.

3 Argument. X shall be of type real.

4 Result Characteristics. Same as X.

5 Result Value. The result has a value equal to a processor-dependent approximation to the error function of X,
\[
\frac{2}{\sqrt{\pi}} \int_{0}^{X} \exp(-t^2) \, dt.
\]

6 Example. ERF (1.0) has the value 0.843 (approximately).

16.9.70 ERFC (X)

1 Description. Complementary error function.

2 Class. Elemental function.

3 Argument. X shall be of type real.

4 Result Characteristics. Same as X.
5 Result Value. The result has a value equal to a processor-dependent approximation to the complementary error function of X, \( 1 - \text{ERF}(X) \); this is equivalent to \( \frac{2}{\sqrt{\pi}} \int_{X}^{\infty} \exp(-t^2) \, dt \).

6 Example. \( \text{ERFC}(1.0) \) has the value 0.157 (approximately).

16.9.71 \textbf{ERFC\_SCALED}(X)

1 Description. Scaled complementary error function.

2 Class. Elemental function.

3 Argument. X shall be of type real.

4 Result Characteristics. Same as X.

5 Result Value. The result has a value equal to a processor-dependent approximation to the exponentially-scaled complementary error function of X, \( \exp(X^2) \frac{2}{\sqrt{\pi}} \int_{X}^{\infty} \exp(-t^2) \, dt \).

6 Example. \( \text{ERFC\_SCALED}(20.0) \) has the value 0.02817434874 (approximately).

\begin{center}
\textbf{NOTE 16.14}

The complementary error function is asymptotic to \( \exp(-X^2)/(X \sqrt{\pi}) \). As such it underflows for \( X \approx 9 \) when using ISO/IEC/IEEE 60559:2011 single precision arithmetic. The exponentially-scaled complementary error function is asymptotic to \( 1/(X \sqrt{\pi}) \). As such it does not underflow until \( X > \text{HUGE}(X)/\sqrt{\pi} \).
\end{center}

16.9.72 \textbf{EVENT\_QUERY}(EVENT, COUNT [, STAT])

1 Description. Query event count.

2 Class. Subroutine.

3 Arguments.

\begin{itemize}
\item EVENT shall be an event variable (16.10.2.10). It shall not be coindexed. It is an INTENT (IN) argument. The EVENT argument is accessed atomically with respect to the execution of EVENT POST statements in unordered segments, in exact analogy to atomic subroutines.
\item COUNT shall be an integer scalar with a decimal range no smaller than that of default integer. It is an INTENT (OUT) argument. If no error condition occurs, COUNT is assigned the value of the count of EVENT; otherwise, it is assigned the value \(-1\).
\item STAT (optional) shall be a noncoindexed integer scalar with a decimal exponent range of at least four. It is an INTENT (OUT) argument. If the STAT argument is present, it is assigned a processor-dependent positive value if an error condition occurs; otherwise it is assigned the value zero. If the STAT argument is not present and an error condition occurs, error termination is initiated.
\end{itemize}

4 Example. If EVENT is an event variable for which there have been no successful posts or waits in preceding segments, and for which there are no posts or waits in an unordered segment, after execution of

\begin{verbatim}
CALL EVENT_QUERY (EVENT, COUNT)
\end{verbatim}

the integer variable COUNT will have the value zero. If there have been ten successful posts to EVENT and two successful waits without an UNTIL\_COUNT= specifier in preceding segments, and for which there are no posts or waits in an unordered segment, after execution of

\begin{verbatim}
CALL EVENT_QUERY (EVENT, COUNT)
\end{verbatim}

the variable COUNT will have the value eight.
NOTE 16.15
Execution of EVENT_QUERY does not imply any synchronization.

16.9.73 EXECUTE_COMMAND_LINE (COMMAND [, WAIT, EXITSTAT, CMDSTAT, CMDMSG ])

1 Description. Execute a command line.
2 Class. Subroutine.
3 Arguments.
   COMMAND shall be a default character scalar. It is an INTENT (IN) argument. Its value is the command line
to be executed. The interpretation is processor dependent.
   WAIT (optional) shall be a logical scalar. It is an INTENT (IN) argument. If WAIT is present with the value
false, and the processor supports asynchronous execution of the command, the command is executed
asynchronously; otherwise it is executed synchronously.
   EXITSTAT (optional) shall be a scalar of type integer with a decimal exponent range of at least nine. It is an
INTENT (INOUT) argument. If the command is executed synchronously, it is assigned the value
of the processor-dependent exit status. Otherwise, the value of EXITSTAT is unchanged.
   CMDSTAT (optional) shall be a scalar of type integer with a decimal exponent range of at least four. It is an
INTENT (OUT) argument. It is assigned the value
−1 if the processor does not support command
line execution, a processor-dependent positive value if an error condition occurs, or the value −2
if no error condition occurs but WAIT is present with the value false and the processor does not
support asynchronous execution. Otherwise it is assigned the value 0.
   CMDMSG (optional) shall be a default character scalar. It is an INTENT (INOUT) argument. If an error condi-
tion occurs, it is assigned a processor-dependent explanatory message. Otherwise, it is unchanged.
4 If the processor supports command line execution, it shall support synchronous and may support asynchronous
execution of the command line.
5 When the command is executed synchronously, EXECUTE_COMMAND_LINE returns after the command line
has completed execution. Otherwise, EXECUTE_COMMAND_LINE returns without waiting.
6 If a condition occurs that would assign a nonzero value to CMDSTAT but the CMDSTAT variable is not present,
error termination is initiated.

16.9.74 EXP (X)
1 Description. Exponential function.
2 Class. Elemental function.
3 Argument. X shall be of type real or complex.
4 Result Characteristics. Same as X.
5 Result Value. The result has a value equal to a processor-dependent approximation to $e^X$. If X is of type
complex, its imaginary part is regarded as a value in radians.
6 Example. EXP (1.0) has the value 2.7182818 (approximately).

16.9.75 EXPONENT (X)
1 Description. Exponent of floating-point number.
2 Class. Elemental function.
Argument. X shall be of type real.

4 Result Characteristics. Default integer.

5 Result Value. The result has a value equal to the exponent \( e \) of the representation for the value of X in the extended real model for the kind of X (16.4), provided X is nonzero and \( e \) is within the range for default integers.

If X has the value zero, the result has the value zero. If X is an IEEE infinity or NaN, the result has the value HUGE (0).

6 Examples. EXPONENT (1.0) has the value 1 and EXPONENT (4.1) has the value 3 for reals whose model is as in Note 16.4.

16.9.76 EXTENDS_TYPE_OF (A, MOLD)

1 Description. Dynamic type extension inquiry.

2 Class. Inquiry function.

3 Arguments.

A shall be an object of extensible declared type or unlimited polymorphic. If it is a polymorphic pointer, it shall not have an undefined association status.

MOLD shall be an object of extensible declared type or unlimited polymorphic. If it is a polymorphic pointer, it shall not have an undefined association status.

4 Result Characteristics. Default logical scalar.

5 Result Value. If MOLD is unlimited polymorphic and is either a disassociated pointer or unallocated allocatable variable, the result is true; otherwise if A is unlimited polymorphic and is either a disassociated pointer or unallocated allocatable variable, the result is false; otherwise if the dynamic type of A or MOLD is extensible, the result is true if and only if the dynamic type of A is an extension type of the dynamic type of MOLD; otherwise the result is processor dependent.

NOTE 16.16

The dynamic type of a disassociated pointer or unallocated allocatable variable is its declared type.

6 Example. Given the declarations and assignments

```plaintext
TYPE T1
  REAL C
END TYPE

TYPE, EXTENDS(T1) :: T2
END TYPE

CLASS(T1), POINTER :: P, Q
ALLOCATE (P)
ALLOCATE (T2 :: Q)
```

the result of EXTENDS_TYPE_OF (P, Q) will be false, and the result of EXTENDS_TYPE_OF (Q, P) will be true.

16.9.77 FAILED_IMAGES([TEAM, KIND])

1 Description. Indices of failed images.

2 Class. Transformational function.

3 Arguments.
TEAM (optional) shall be a scalar of type TEAM_TYPE from the intrinsic module ISO_FORTRAN_ENV. Its value shall be that of the current or an ancestor team. If TEAM is absent, the team specified is the current team.

KIND (optional) shall be a scalar integer constant expression.

4 Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default integer type. The result is an array of rank one whose size is equal to the number of images in the specified team that are known by the invoking image to have failed.

5 Result Value. The elements of the result are the values of the image indices of the known failed images in the specified team, in numerically increasing order. If the executing image has previously executed an image control statement whose STAT= specifier assigned the value STAT_FAILED_IMAGE from the intrinsic module ISO_FORTRAN_ENV, or referenced a collective subroutine whose STAT argument was set to STAT_FAILED_IMAGE, at least one image in the set of images participating in that image control statement or collective subroutine reference shall be known to have failed.

6 Examples. If image 3 is the only image in the current team that is known by the invoking image to have failed, FAILED_IMAGES() will have the value [3]. If there are no images in the current team that are known by the invoking image to have failed, the value of FAILED_IMAGES() will be a zero-sized array.

16.9.78 FINDLOC (ARRAY, VALUE, DIM [, MASK, KIND, BACK]) or FINDLOC (ARRAY, VALUE [, MASK, KIND, BACK])

1 Description. Location(s) of a specified value.

2 Class. Transformational function.

3 Arguments.

ARRAY shall be an array of intrinsic type.

VALUE shall be scalar and in type conformance with ARRAY, as specified in Table 10.2 for the operator == or the operator .EQV..

DIM shall be an integer scalar with a value in the range \(1 \leq DIM \leq n\), where \(n\) is the rank of ARRAY.

MASK (optional) shall be of type logical and shall be conformable with ARRAY.

KIND (optional) shall be a scalar integer constant expression.

BACK (optional) shall be a logical scalar.

4 Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type. If DIM does not appear, the result is an array of rank one and of size equal to the rank of ARRAY; otherwise, the result is of rank \(n-1\) and shape \([d_1, d_2, \ldots, d_{DIM-1}, d_{DIM+1}, \ldots, d_n]\), where \([d_1, d_2, \ldots, d_n]\) is the shape of ARRAY.

5 Result Value.

Case (i): The result of FINDLOC (ARRAY, VALUE) is a rank-one array whose element values are the values of the subscripts of an element of ARRAY whose value matches VALUE. If there is such a value, the \(i^{th}\) element value is in the range 1 to \(e_i\), where \(e_i\) is the extent of the \(i^{th}\) dimension of ARRAY. If no elements match VALUE or ARRAY has size zero, all elements of the result are zero.

Case (ii): The result of FINDLOC (ARRAY, VALUE, MASK = MASK) is a rank-one array whose element values are the values of the subscripts of an element of ARRAY, corresponding to a true element of MASK, whose value matches VALUE. If there is such a value, the \(i^{th}\) element value is in the range 1 to \(e_i\), where \(e_i\) is the extent of the \(i^{th}\) dimension of ARRAY. If no elements match VALUE, ARRAY has size zero, or every element of MASK has the value false, all elements of the result are zero.

Case (iii): If ARRAY has rank one, the result of
FINDLOC (ARRAY, VALUE, DIM=DIM [, MASK = MASK]) is a scalar whose value is equal to that of the first element of FINDLOC (ARRAY, VALUE [, MASK = MASK]). Otherwise, the value of element $(s_1, s_2, \ldots, s_{DIM-1}, s_{DIM+1}, \ldots, s_n)$ of the result is equal to FINDLOC (ARRAY $(s_1, s_2, \ldots, s_{DIM-1}, s_{DIM+1}, \ldots, s_n)$, VALUE, DIM=1 [, MASK = MASK $(s_1, s_2, \ldots, s_{DIM-1}, \ldots, s_n)$]).

If both ARRAY and VALUE are of type logical, the comparison is performed with the .EQV. operator; otherwise, the comparison is performed with the == operator. If the value of the comparison is true, that element of ARRAY matches VALUE.

If DIM is not present, more than one element matches VALUE, and BACK is absent or present with the value false, the value returned indicates the first such element, taken in array element order. If DIM is not present and BACK is present with the value true, the value returned indicates the last such element, taken in array element order.

Examples.

Case (i): The value of FINDLOC ([2, 6, 4, 6], VALUE=6) is [2], and the value of FINDLOC ([2, 6, 4, 6], VALUE=6, BACK=.TRUE.) is [4].

Case (ii): If A has the value \[
\begin{bmatrix}
0 & -5 & 7 & 7 \\
3 & 4 & -1 & 2 \\
1 & 5 & 6 & 7
\end{bmatrix},
\]
and M has the value \[
\begin{bmatrix}
T & T & F & T \\
T & T & F & T \\
T & T & F & T
\end{bmatrix},
\]
FINDLOC (A, 7, MASK = M) has the value [1, 4] and FINDLOC (A, 7, MASK = M, BACK = .TRUE.) has the value [3, 4]. This is independent of the declared lower bounds for A.

Case (iii): The value of FINDLOC ([2, 6, 4], VALUE=6, DIM=1) is 2. If B has the value \[
\begin{bmatrix}
1 & 2 & -9 \\
2 & 2 & 6
\end{bmatrix},
\]
FINDLOC (B, VALUE=2, DIM=1) has the value [2, 1, 0] and FINDLOC (B, VALUE=2, DIM=2) has the value [2, 1]. This is independent of the declared lower bounds for B.

16.9.79 FLOOR (A [, KIND])

Description. Greatest integer less than or equal to A.

Class. Elemental function.

Arguments. A shall be of type real. KIND (optional) shall be a scalar integer constant expression.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default integer type.

Result Value. The result has a value equal to the greatest integer less than or equal to A.

Examples. FLOOR (3.7) has the value 3. FLOOR (−3.7) has the value −4.

16.9.80 FRACTION (X)

Description. Fractional part of number.

Class. Elemental function.

Argument. X shall be of type real.

Result Characteristics. Same as X.

Result Value. The result has the value $X \times b^{-e}$, where $b$ and $e$ are as defined in 16.4 for the representation of X in the extended real model for the kind of X. If X has the value zero, the result is zero. If X is an IEEE NaN, the result is that NaN. If X is an IEEE infinity, the result is an IEEE NaN.
Example. FRACTION (3.0) has the value 0.75 for reals whose model is as in Note 16.4.

16.9.81 GAMMA (X)

1 Description. Gamma function.

2 Class. Elemental function.

3 Argument. X shall be of type real. Its value shall not be a negative integer or zero.

4 Result Characteristics. Same as X.

5 Result Value. The result has a value equal to a processor-dependent approximation to the gamma function of X,

\[ \Gamma(X) = \begin{cases} \int_0^\infty t^{X-1} \exp(-t) \, dt & X > 0 \\ \int_0^\infty t^{X-1} \left( \exp(-t) - \sum_{k=0}^n \frac{(-t)^k}{k!} \right) \, dt & -n - 1 < X < -n, \text{n an integer } \geq 0 \end{cases} \]

6 Example. GAMMA (1.0) has the value 1.000 (approximately).

16.9.82 GET_COMMAND ([COMMAND, LENGTH, STATUS, ERRMSG])

1 Description. Get program invocation command.

2 Class. Subroutine.

3 Arguments. COMMAND (optional) shall be a default character scalar. It is an INTENT (OUT) argument. It is assigned the entire command by which the program was invoked. If the command cannot be determined, COMMAND is assigned all blanks.

LENGTH (optional) shall be a scalar of type integer with a decimal exponent range of at least four. It is an INTENT (OUT) argument. It is assigned the significant length of the command by which the program was invoked. The significant length may include trailing blanks if the processor allows commands with significant trailing blanks. This length does not consider any possible truncation or padding in assigning the command to the COMMAND argument; in fact the COMMAND argument need not even be present. If the command length cannot be determined, a length of 0 is assigned.

STATUS (optional) shall be a scalar of type integer with a decimal exponent range of at least four. It is an INTENT (OUT) argument. It is assigned the value \(-1\) if the COMMAND argument is present and has a length less than the significant length of the command. It is assigned a processor-dependent positive value if the command retrieval fails. Otherwise it is assigned the value 0.

ERRMSG (optional) shall be a default character scalar. It is an INTENT (INOUT) argument. It is assigned a processor-dependent explanatory message if the command retrieval fails. Otherwise, it is unchanged.

4 Example. If the program below is invoked with the command “example” on a processor that supports command retrieval, it will display “Hello example”.

```fortran
PROGRAM hello
   CHARACTER(:), ALLOCATABLE :: cmd
   INTEGER :: cmdlen
   CALL GET_COMMAND(LENGTH=cmdlen)
   IF (cmdlen>0) THEN
      ALLOCATE(CHARACTER(cmdlen) :: cmd)
      CALL GET_COMMAND(cmd)
```

ISO/IEC JTC 1/SC 22/WG5/N2118
PRINT *, 'Hello ', cmd
END IF
END PROGRAM

16.9.83 GET_COMMAND_ARGUMENT (NUMBER [, VALUE, LENGTH, STATUS, ERRMSG])

1 Description. Get program invocation argument.

2 Class. Subroutine.

3 Arguments.

4 NUMBER shall be an integer scalar. It is an INTENT (IN) argument that specifies the number of the command argument that the other arguments give information about.

5 Command argument 0 always exists, and is the command name by which the program was invoked if the processor has such a concept; otherwise, the value of command argument 0 is processor dependent. The remaining command arguments are numbered consecutively from 1 to the argument count in an order determined by the processor.

6 VALUE (optional) shall be a default character scalar. It is an INTENT (OUT) argument. If the command argument specified by NUMBER exists, its value is assigned to VALUE; otherwise, VALUE is assigned all blanks.

7 LENGTH (optional) shall be a scalar of type integer with a decimal exponent range of at least four. It is an INTENT (OUT) argument. If the command argument specified by NUMBER exists, its significant length is assigned to LENGTH; otherwise, LENGTH is assigned the value zero. It is processor dependent whether the significant length includes trailing blanks. This length does not consider any possible truncation or padding in assigning the command argument value to the VALUE argument; in fact the VALUE argument need not even be present.

8 STATUS (optional) shall be a scalar of type integer with a decimal exponent range of at least four. It is an INTENT (OUT) argument. If NUMBER is less than zero or greater than the argument count that would be returned by the intrinsic function COMMAND_ARGUMENT_COUNT, or command retrieval fails, STATUS is assigned a processor-dependent positive value. Otherwise, if VALUE is present and has a length less than the significant length of the specified command argument, it is assigned the value $-1$. Otherwise it is assigned the value 0.

9 ERRMSG (optional) shall be a default character scalar. It is an INTENT (INOUT) argument. It is assigned a processor-dependent explanatory message if the optional argument STATUS is, or would be if present, assigned a positive value. Otherwise, it is unchanged.

4 Example. On a processor that supports command arguments, the following program displays the arguments of the command by which it was invoked.

PROGRAM show_arguments
  INTEGER :: i
  CHARACTER :: command*32, arg*128
  CALL get_command_argument(0, command)
  WRITE (*,*) "Command name is: ", command
  DO i = 1, command_argument_count()
    CALL get_command_argument(i, arg)
    WRITE (*,*) "Argument ", i, " is ", arg
  END DO
END PROGRAM show_arguments
16.9.84  \texttt{GET\_ENVIRONMENT\_VARIABLE (NAME [, VALUE, LENGTH, STATUS, TRIM\_NAME, ERRMSG])}

\begin{verbatim}
1 Description. Get environment variable.
2 Class. Subroutine.
3 Arguments.
4 \texttt{NAME} shall be a default character scalar. It is an \texttt{INTENT (IN)} argument. The interpretation of case is processor dependent.
5 \texttt{VALUE} (optional) shall be a default character scalar. It is an \texttt{INTENT (OUT)} argument. It is assigned the value of the environment variable specified by \texttt{NAME}. \texttt{VALUE} is assigned all blanks if the environment variable does not exist or does not have a value or if the processor does not support environment variables.
6 \texttt{LENGTH} (optional) shall be a scalar of type integer with a decimal exponent range of at least four. It is an \texttt{INTENT (OUT)} argument. If the specified environment variable exists and has a value, \texttt{LENGTH} is assigned the value of its length. Otherwise \texttt{LENGTH} is assigned the value zero.
7 \texttt{STATUS} (optional) shall be a scalar of type integer with a decimal exponent range of at least four. It is an \texttt{INTENT (OUT)} argument. If the environment variable exists and either has no value, its value is successfully assigned to \texttt{VALUE}, or the \texttt{VALUE} argument is not present, \texttt{STATUS} is assigned the value zero. \texttt{STATUS} is assigned the value $-1$ if the \texttt{VALUE} argument is present and has a length less than the significant length of the environment variable. It is assigned the value 1 if the specified environment variable does not exist, or 2 if the processor does not support environment variables. Processor-dependent values greater than 2 may be assigned for other error conditions.
8 \texttt{TRIM\_NAME} (optional) shall be a logical scalar. It is an \texttt{INTENT (IN)} argument. If \texttt{TRIM\_NAME} is present with the value false then trailing blanks in \texttt{NAME} are considered significant if the processor supports trailing blanks in environment variable names. Otherwise trailing blanks in \texttt{NAME} are not considered part of the environment variable's name.
9 \texttt{ERRMSG} (optional) shall be a default character scalar. It is an \texttt{INTENT (INOUT)} argument. It is assigned a processor-dependent explanatory message if the optional argument \texttt{STATUS} is, or would be if present, assigned a positive value. Otherwise, it is unchanged.
\end{verbatim}

4 It is processor dependent whether an environment variable that exists on an image also exists on another image, and if it does exist on both images, whether the values are the same or different.

5 Example. If the value of the environment variable \texttt{DATAFILE} is datafile.dat, executing the statement sequence below will assign the value 'datafile.dat' to \texttt{FILENAME}.

\begin{verbatim}
CHARACTER(:), ALLOCATABLE :: FILENAME
INTEGER :: NAMELEN
CALL GET\_ENVIRONMENT\_VARIABLE ("DATAFILE", LENGTH=NAMELEN)
IF (LENGTH>0) THEN
  ALLOCATE(CHARACTER(LENGTH) :: FILENAME)
  CALL GET\_ENVIRONMENT\_VARIABLE("DATAFILE", FILENAME)
END IF
\end{verbatim}

16.9.85  \texttt{GET\_TEAM([LEVEL])}

\begin{verbatim}
1 Description. Team.
2 Class. Transformational function.
3 Argument. \texttt{LEVEL} (optional) shall be a scalar integer whose value shall be equal to one of the named constants \texttt{INITIAL\_TEAM, PARENT\_TEAM, or CURRENT\_TEAM} from the intrinsic module \texttt{ISO\_FORTRAN\_ENV}.
\end{verbatim}
**Result Characteristics.** Scalar of type TEAM_TYPE from the intrinsic module ISO_FORTRAN_ENV.

**Result Value.** The result is a TEAM_TYPE value specifying the current team if LEVEL is not present, present with the value CURRENT_TEAM, or the current team is the initial team. Otherwise, the result specifies the parent team if LEVEL is present with the value PARENT_TEAM, and specifies the initial team if LEVEL is present with the value INITIAL_TEAM.

**Examples.**

```fortran
PROGRAM EXAMPLE1
    USE, INTRINSIC :: ISO_FORTRAN_ENV, ONLY: TEAM_TYPE
    TYPE(TEAM_TYPE) :: WORLD_TEAM, TEAM2

    ! Define a team variable representing the initial team
    WORLD_TEAM = GET_TEAM()

END PROGRAM

SUBROUTINE EXAMPLE2 (A)
    USE, INTRINSIC :: ISO_FORTRAN_ENV, ONLY: TEAM_TYPE
    REAL A[*]
    TYPE(TEAM_TYPE) :: NEW_TEAM, PARENT_TEAM

    ! Form NEW_TEAM

    PARENT_TEAM = GET_TEAM()

    CHANGE TEAM (NEW_TEAM)

    ! Reference image 1 in parent’s team
    A [1, TEAM=PARENT_TEAM] = 4.2

    ! Reference image 1 in current team

END TEAM
END SUBROUTINE TT
```

**16.9.86 HUGE (X)**

**Description.** Largest model number.

**Class.** Inquiry function.

**Argument.** X shall be of type integer or real. It may be a scalar or an array.

**Result Characteristics.** Scalar of the same type and kind type parameter as X.

**Result Value.** The result has the value $r^q - 1$ if X is of type integer and $(1 - b^{-p})b^{e_{\text{max}}}$ if X is of type real, where r, q, b, p, and $e_{\text{max}}$ are as defined in 16.4 for the model representing numbers of the same type and kind type parameter as X.

**Example.** HUGE (X) has the value $(1 - 2^{-24}) \times 2^{127}$ for real X whose model is as in Note 16.4.
16.9.87  **HYPOT (X, Y)**

**Description.** Euclidean distance function.

**Class.** Elemental function.

**Arguments.**

- X shall be of type real.
- Y shall be of type real with the same kind type parameter as X.

**Result Characteristics.** Same as X.

**Result Value.** The result has a value equal to a processor-dependent approximation to the Euclidean distance, $\sqrt{X^2 + Y^2}$, without undue overflow or underflow.

**Example.** HYPOT (3.0, 4.0) has the value 5.0 (approximately).

16.9.88  **IACHAR (C [, KIND])**

**Description.** ASCII code value for character.

**Class.** Elemental function.

**Arguments.**

- C shall be of type character and of length one.
- KIND (optional) shall be a scalar integer constant expression.

**Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default integer type.

**Result Value.** If C is in the collating sequence defined by the codes specified in ISO/IEC 646:1991 (International Reference Version), the result is the position of C in that sequence; it is nonnegative and less than or equal to 127. The value of the result is processor dependent if C is not in the ASCII collating sequence. The results are consistent with the LGE, LGT, LLE, and LLT comparison functions. For example, if LLE (C, D) is true, IACHAR (C) <= IACHAR (D) is true where C and D are any two characters representable by the processor.

**Example.** IACHAR ('X') has the value 88.

16.9.89  **IALL (ARRAY, DIM [, MASK]) or IALL (ARRAY [, MASK])**

**Description.** Array reduced by IAND function.

**Class.** Transformational function.

**Arguments.**

- ARRAY shall be an array of type integer.
- DIM shall be an integer scalar with a value in the range $1 \leq DIM \leq n$, where $n$ is the rank of ARRAY.
- MASK (optional) shall be of type logical and shall be conformable with ARRAY.

**Result Characteristics.** The result is of the same type and kind type parameter as ARRAY. It is scalar if DIM does not appear or if ARRAY has rank one; otherwise, the result is an array of rank $n - 1$ and shape $[d_1, d_2, \ldots, d_{DIM-1}, d_{DIM+1}, \ldots, d_n]$ where $[d_1, d_2, \ldots, d_n]$ is the shape of ARRAY.

**Result Value.**

1. If ARRAY has size zero the result value is equal to NOT (INT (0, KIND (ARRAY))). Otherwise, the result of IALL (ARRAY) has a value equal to the bitwise AND of all the elements of ARRAY.
**Case (ii):** The result of `IALL (ARRAY, MASK=MASK)` has a value equal to `IALL (PACK (ARRAY, MASK))`.

**Case (iii):** The result of `IALL (ARRAY, DIM=DIM [, MASK=MASK])` has a value equal to that of `IALL (ARRAY [, MASK=MASK])` if `ARRAY` has rank one. Otherwise, the value of element \((s_1, s_2, \ldots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \ldots, s_n)\) of the result is equal to `IALL (ARRAY (s_1, s_2, \ldots, s_{\text{DIM}-1}, : , s_{\text{DIM}+1}, \ldots, s_n) [, \text{MASK} = \text{MASK} (s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n)])`.

**Examples.** `IALL ([14, 13, 11])` has the value 8. `IALL ([14, 13, 11], \text{MASK}=[.true., .false., .true])` has the value 10.

**16.9.90 IAND (I, J)**

**Description.** Bitwise AND.

**Class.** Elemental function.

**Arguments.**
- `I` shall be of type integer or a `boz-literal-constant`.
- `J` shall be of type integer or a `boz-literal-constant`. If both `I` and `J` are of type integer, they shall have the same kind type parameter. `I` and `J` shall not both be `boz-literal-constants`.

**Result Characteristics.** Same as `I` if `I` is of type integer; otherwise, same as `J`.

**Result Value.** If either `I` or `J` is a `boz-literal-constant`, it is first converted as if by the intrinsic function `INT` to type integer with the kind type parameter of the other. 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>
<th>J</th>
<th>IAND (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 model for the interpretation of an integer value as a sequence of bits is in 16.3.

**Example.** `IAND (1, 3)` has the value 1.

**16.9.91 IANY (ARRAY, DIM [, MASK]) or IANY (ARRAY [, MASK])**

**Description.** Reduce array with bitwise OR operation.

**Class.** Transformational function.

**Arguments.**
- `ARRAY` shall be of type integer. It shall be an array.
- `DIM` shall be an integer scalar with a value in the range \(1 \leq \text{DIM} \leq n\), where \(n\) is the rank of `ARRAY`.
- `MASK` (optional) shall be of type logical and shall be conformable with `ARRAY`.

**Result Characteristics.** The result is of the same type and kind type parameter as `ARRAY`. It is scalar if `DIM` does not appear or if `ARRAY` has rank one; otherwise, the result is an array of rank \(n-1\) and shape \([d_1, d_2, \ldots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \ldots, d_n]\) where \([d_1, d_2, \ldots, d_n]\) is the shape of `ARRAY`.

**Result Value.**
- **Case (i):** The result of `IANY (ARRAY)` is the bitwise OR of all the elements of `ARRAY`. If `ARRAY` has size zero the result value is equal to zero.
Case (ii): The result of \( \text{ANY} \) (\text{ARRAY}, \text{MASK} = \text{MASK}) has a value equal to \( \text{ANY} \) (\text{PACK} (\text{ARRAY}, \text{MASK})).

Case (iii): The result of \( \text{ANY} \) (\text{ARRAY}, \text{DIM} = \text{DIM}[, \text{MASK} = \text{MASK}]) has a value equal to that of \( \text{ANY} \) (\text{ARRAY}[, \text{MASK} = \text{MASK}]) if \text{ARRAY} has rank one. Otherwise, the value of element \((s_1, s_2, \ldots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \ldots, s_n)\) of the result is equal to \( \text{ANY} \) (\text{ARRAY} (s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n[, \text{MASK} = \text{MASK} (s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n)]).

Examples. \( \text{ANY} ([14, 13, 8]) \) has the value 15. \( \text{ANY} ([14, 13, 8], \text{MASK} = [\text{.true., .false., .true.}]) \) has the value 14.

### 16.9.92 IBCLR (I, POS)

**Description.** I with bit POS replaced by zero.

**Class.** Elemental function.

**Arguments.**
- I shall be of type integer.
- POS shall be of type integer. It shall be nonnegative and less than BIT\_SIZE (I).

**Result Characteristics.** Same as I.

**Result Value.** The result has the value of the sequence of bits of I, except that bit POS is zero. The model for the interpretation of an integer value as a sequence of bits is in 16.3.

**Examples.** IBCLR (14, 1) has the value 12. If V has the value [1, 2, 3, 4], the value of IBCLR (POS = V, I = 31) is [29, 27, 23, 15].

### 16.9.93 IBITS (I, POS, LEN)

**Description.** Specified sequence of bits.

**Class.** Elemental function.

**Arguments.**
- I shall be of type integer.
- POS shall be of type integer. It shall be nonnegative and POS + LEN shall be less than or equal to BIT\_SIZE (I).
- LEN shall be of type integer and nonnegative.

**Result Characteristics.** 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 model for the interpretation of an integer value as a sequence of bits is in 16.3.

**Example.** IBITS (14, 1, 3) has the value 7.

### 16.9.94 IBSET (I, POS)

**Description.** I with bit POS replaced by one.

**Class.** Elemental function.

**Arguments.**
- I shall be of type integer.
- POS shall be of type integer. It shall be nonnegative and less than BIT\_SIZE (I).
Result Characteristics. Same as I.

Result Value. The result has the value of the sequence of bits of I, except that bit POS is one. The model for the interpretation of an integer value as a sequence of bits is in 16.3.

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].

16.9.95 ICHAR (C [, KIND])

Description. Code value for character.

Class. Elemental function.

Arguments.

C shall be of type character and of length one. Its value shall be that of a character capable of representation in the processor.

KIND (optional) shall be a scalar integer constant expression.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default integer type.

Result Value. The result is the position of C in the processor collating sequence associated with the kind type parameter of C; it is nonnegative and less than \( n \), where \( n \) is the number of characters in the collating sequence. The kind type parameter of the result shall specify an integer kind that is capable of representing \( n \). For any characters C and D capable of representation in the processor, \( C \leq D \) is true if and only if ICHAR (C) \( \leq \) ICHAR (D) is true and \( C = D \) is true if and only if ICHAR (C) \( = \) ICHAR (D) is true.

Example. ICHAR (’X’) has the value 88 on a processor using the ASCII collating sequence for default characters.

16.9.96 IEOR (I, J)

Description. Bitwise exclusive OR.

Class. Elemental function.

Arguments.

I shall be of type integer or a boz-literal-constant.

J shall be of type integer or a boz-literal-constant. If both I and J are of type integer, they shall have the same kind type parameter. I and J shall not both be boz-literal-constants.

Result Characteristics. Same as I if I is of type integer; otherwise, same as J.

Result Value. If either I or J is a boz-literal-constant, it is first converted as if by the intrinsic function INT to type integer with the kind type parameter of the other. 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>
<th>J</th>
<th>IEOR (I, J)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The model for the interpretation of an integer value as a sequence of bits is in 16.3.

Example. IEOR (1, 3) has the value 2.
16.9.97  IMAGE_INDEX (COARRAY, SUB) or (COARRAY, SUB, TEAM) or (COARRAY, SUB, TEAM_NUMBER)

1 Description. Image index from cosubscripts.

2 Class. Inquiry function.

3 Arguments.
   COARRAY shall be a coarray of any type. If TEAM_NUMBER appears and the current team is not the initial team, it shall be established in an ancestor of the current team. Otherwise, if TEAM appears, it shall be established in that team. Otherwise, it shall be established in the current team.

   SUB shall be a rank-one integer array of size equal to the corank of COARRAY.

   TEAM shall be a scalar of type TEAM_TYPE from the intrinsic module ISO_FORTRAN_ENV, with a value that identifies the current or an ancestor team.

   TEAM_NUMBER shall be a positive integer scalar. If the current team is not the initial team, it shall identify a team whose parent is the same as that of the current team.

4 Result Characteristics. Default integer scalar.

5 Result Value. If the value of SUB is a valid sequence of cosubscripts for COARRAY in the team specified by TEAM or TEAM_NUMBER, or the current team if neither TEAM nor TEAM_NUMBER appears, the result is the index of the corresponding image in that team. Otherwise, the result is zero.

6 Examples. If A and B are declared as A [0:*] and B (10, 20) [10, 0:9, 0:*] respectively, IMAGE_INDEX (A, [0]) has the value 1 and IMAGE_INDEX (B, [3, 1, 2]) has the value 213 (on any image).

16.9.98  IMAGE_STATUS (IMAGE [, TEAM])

1 Description. Image execution state.

2 Class. Elemental function.

3 Arguments.
   IMAGE shall be of type integer. Its value shall be positive and less than or equal to the number of images in the specified team.

   TEAM (optional) shall be a scalar of type TEAM_TYPE from the intrinsic module ISO_FORTRAN_ENV. Its value shall represent the current or an ancestor team. If TEAM is absent, the team specified is the current team.

4 Result Characteristics. Default integer.

5 Result Value. The result value is STAT_FAILED_IMAGE from the intrinsic module ISO_FORTRAN_ENV if the specified image has failed, STAT_STOPPED_IMAGE from the intrinsic module ISO_FORTRAN_ENV if that image has initiated normal termination, and zero otherwise.

6 Example. If image 3 of the current team has failed, IMAGE_STATUS (3) has the value STAT_FAILED_IMAGE.

16.9.99  INDEX (STRING, SUBSTRING [, BACK, KIND])

1 Description. Character string search.

2 Class. Elemental function.

3 Arguments.
   STRING shall be of type character.
SUBSTRING shall be of type character with the same kind type parameter as STRING.
BACK (optional) shall be of type logical.
KIND (optional) shall be a scalar integer constant expression.

4 Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type.

5 Result Value.

Case (i): If STRING % LEN < SUBSTRING % LEN, the result has the value zero.
Case (ii): Otherwise, if there is an integer I in the range 1 ≤ I ≤ STRING % LEN − SUBSTRING % LEN + 1, such that STRING(I : I + SUBSTRING % LEN − 1) is equal to SUBSTRING, the result has the value of the smallest such I if BACK is absent or present with the value false, and the greatest such I if BACK is present with the value true.
Case (iii): Otherwise, the result has the value zero.

6 Examples. INDEX ('FORTRAN', 'R') has the value 3. INDEX ('FORTRAN', 'R', BACK = .TRUE.) has the value 5.

16.9.100 INT (A [, KIND])

1 Description. Conversion to integer type.

2 Class. Elemental function.

3 Arguments.
A shall be of type integer, real, or complex, or a boz-literal-constant.
KIND (optional) shall be a scalar integer constant expression.

4 Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default integer type.

5 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| ≥ 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) = INT (REAL (A, KIND (A))).
Case (iv): If A is a boz-literal-constant, the value of the result is the value whose bit sequence according to the model in 16.3 is the same as that of A as modified by padding or truncation according to 16.3.3. The interpretation of a bit sequence whose most significant bit is 1 is processor dependent.

6 Example. INT (−3.7) has the value −3.

16.9.101 IOR (I, J)

1 Description. Bitwise inclusive OR.

2 Class. Elemental function.

3 Arguments.
I shall be of type integer or a boz-literal-constant.
J shall be of type integer or a boz-literal-constant. If both I and J are of type integer, they shall have the same kind type parameter. I and J shall not both be boz-literal-constants.

4 Result Characteristics. Same as I if I is of type integer; otherwise, same as J.
5 Result Characteristics. Same as I.

6 Result Value. If either I or J is a boz-literal-constant, it is first converted as if by the intrinsic function INT to

bit integer with the kind type parameter of the other. 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>
<th>J</th>
<th>IOR (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>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

7 The model for the interpretation of an integer value as a sequence of bits is in 16.3.

8 Example. IOR (5, 3) has the value 7.

16.9.102 IPARITY (ARRAY, DIM [, MASK]) or IPARITY (ARRAY [, MASK])

1 Description. Array reduced by IEOR function.

2 Class. Transformational function.

3 Arguments.

   ARRAY shall be of type integer. It shall be an array.

   DIM shall be an integer scalar with a value in the range \(1 \leq \text{DIM} \leq n\), where \(n\) is the rank of ARRAY.

   MASK (optional) shall be of type logical and shall be conformable with ARRAY.

4 Result Characteristics. The result is of the same type and kind type parameter as ARRAY. It is scalar if

DIM does not appear; otherwise, the result has rank \(n-1\) and shape \([d_1, d_2, \ldots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \ldots, d_n]\) where

\([d_1, d_2, \ldots, d_n]\) is the shape of ARRAY.

5 Result Value.

   Case (i): The result of IPARITY (ARRAY) has a value equal to the bitwise exclusive OR of all the elements

   of ARRAY. If ARRAY has size zero the result has the value zero.

   Case (ii): The result of IPARITY (ARRAY, MASK=MASK) has a value equal to that of IPARITY (PACK

   ARRAY, MASK).

   Case (iii): The result of IPARITY (ARRAY, DIM=\text{DIM} [, MASK=MASK]) has a value equal to that of

   IPARITY (ARRAY [, MASK=MASK]) if ARRAY has rank one. Otherwise, the value of element

   \((s_1, s_2, \ldots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \ldots, s_n)\) of the result is equal to IPARITY (ARRAY \((s_1, s_2, \ldots, s_{\text{DIM}-1}, \ldots, s_n)\) [, MASK = MASK \((s_1, s_2, \ldots, s_{\text{DIM}-1}, \ldots, s_n)\)].

6 Examples. IPARITY ([14, 13, 8]) has the value 11. IPARITY ([14, 13, 8], MASK=[true,, false,, true]) has the

value 6.

16.9.103 ISHFT (I, SHIFT)

1 Description. Logical shift.

2 Class. Elemental function.

3 Arguments.

   I shall be of type integer.

   SHIFT shall be of type integer. The absolute value of SHIFT shall be less than or equal to BIT_SIZE (I).

4 Result Characteristics. 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; if SHIFT is zero, no shift is performed. Bits shifted out from the left or from the right, as appropriate, are lost. Zeros are shifted in from the opposite end. The model for the interpretation of an integer value as a sequence of bits is in 16.3.

Example. ISHFT (3, 1) has the value 6.

16.9.104 ISHFTC (I, SHIFT [, SIZE])

Description. Circular shift of the rightmost bits.

Class. Elemental function.

Arguments.

I shall be of type integer.

SHIFT shall be of type integer. The absolute value of SHIFT shall be less than or equal to SIZE.

SIZE (optional) shall be of type integer. The value of SIZE shall be positive and shall not exceed BIT_SIZE (I).

If SIZE is absent, it is as if it were present with the value of BIT_SIZE (I).

Result Characteristics. 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 model for the interpretation of an integer value as a sequence of bits is in 16.3.

Example. ISHFTC (3, 2, 3) has the value 5.

16.9.105 IS_CONTIGUOUS (ARRAY)

Description. Array contiguity test (8.5.7).

Class. Inquiry function.

Argument. ARRAY may be of any type. It shall be assumed-rank or an array. If it is a pointer it shall be associated.

Result Characteristics. Default logical scalar.

Result Value. The result has the value true if ARRAY has rank zero or is contiguous, and false otherwise.

Example. After the pointer assignment AP = TARGET (1:10:2), IS_CONTIGUOUS (AP) has the value false.

16.9.106 IS_IOSTAT_END (I)

Description. IOSTAT value test for end of file.

Class. Elemental function.

Argument. I shall be of type integer.

Result Characteristics. Default logical.

Result Value. The result has the value true if and only if I is a value for the scalar-int-variable in an IOSTAT= specifier (12.11.5) that would indicate an end-of-file condition.
16.9.107  IS_IOSTAT_EOR (I)

Description. IOSTAT value test for end of record.

Class. Elemental function.

Argument. I shall be of type integer.

Result Characteristics. Default logical.

Result Value. The result has the value true if and only if I is a value for the scalar-int-variable in an IOSTAT=
specifier (12.11.5) that would indicate an end-of-record condition.

16.9.108  KIND (X)

Description. Value of the kind type parameter of X.

Class. Inquiry function.

Argument. X may be of any intrinsic type. It may be a scalar or an array.

Result Characteristics. Default integer scalar.

Result Value. The result has a value equal to the kind type parameter value of X.

Example. KIND (0.0) has the kind type parameter value of default real.

16.9.109  LBOUND (ARRAY [, DIM, KIND])

Description. Lower bound(s).

Class. Inquiry function.

Arguments.

ARRAY shall be assumed-rank or an array. It shall not be an unallocated allocatable variable or a pointer
that is not associated.

DIM (optional) shall be an integer scalar with a value in the range $1 \leq \text{DIM} \leq n$, where $n$ is the rank of ARRAY.

The corresponding actual argument shall not be an optional dummy argument, a disassociated
pointer, or an unallocated allocatable.

KIND (optional) shall be a scalar integer constant expression.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of
KIND; otherwise the kind type parameter is that of default integer type. The result is scalar if DIM is present;
otherwise, the result is an array of rank one and size $n$, where $n$ is the rank of ARRAY.

Result Value.

Case (i): If DIM is present, ARRAY is a whole array, and either ARRAY is an assumed-size array of rank
DIM or dimension DIM of ARRAY has nonzero extent, the result has a value equal to the lower
bound for subscript DIM of ARRAY. Otherwise, if DIM is present, the result value is 1.

Case (ii): LBOUND (ARRAY) has a value whose $i^{th}$ element is equal to LBOUND (ARRAY, $i$), for $i = 1, 2, 
\ldots, n$, where $n$ is the rank of ARRAY. LBOUND (ARRAY, KIND=KIND) has a value whose $i^{th}$
element is equal to LBOUND (ARRAY, $i$, KIND), for $i = 1, 2, \ldots, n$, where $n$ is the rank of
ARRAY.

NOTE 16.17

If ARRAY is assumed-rank and has rank zero, DIM cannot be present since it cannot satisfy the requirement $1 \leq \text{DIM} \leq 0$. 398
Examples. If A is declared by the statement

```fortran
REAL A (2:3, 7:10)
```

then LBOUND (A) is [2, 7] and LBOUND (A, DIM=2) is 7.

16.9.110 LCOBOUND (COARRAY [, DIM, KIND])

Description. Lower cobound(s) of a coarray.

Class. Inquiry function.

Arguments.

COARRAY shall be a coarray and may be of any type. It may be a scalar or an array. If it is allocatable it shall be allocated.

DIM (optional) shall be an integer scalar with a value in the range \(1 \leq DIM \leq n\), where \(n\) is the corank of COARRAY. The corresponding actual argument shall not be an optional dummy argument, a disassociated pointer, or an unallocated allocatable.

KIND (optional) shall be a scalar integer constant expression.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default integer type. The result is scalar if DIM is present; otherwise, the result is an array of rank one and size \(n\), where \(n\) is the corank of COARRAY.

Result Value.

Case (i): If DIM is present, the result has a value equal to the lower cobound for codimension DIM of COARRAY.

Case (ii): If DIM is absent, the result has a value whose \(i^{th}\) element is equal to the lower cobound for codimension \(i\) of COARRAY, for \(i = 1, 2, \ldots, n\), where \(n\) is the corank of COARRAY.

Examples. If A is allocated by the statement ALLOCATE (A [2:3, 7:*]) then LCOBOUND (A) is [2, 7] and LCOBOUND (A, DIM=2) is 7.

16.9.111 LEADZ (I)

Description. Number of leading zero bits.

Class. Elemental function.

Argument. I shall be of type integer.

Result Characteristics. Default integer.

Result Value. If all of the bits of I are zero, the result has the value BIT_SIZE (I). Otherwise, the result has the value BIT_SIZE (I) – 1 – \(k\), where \(k\) is the position of the leftmost 1 bit in I. The model for the interpretation of an integer value as a sequence of bits is in 16.3.

Examples. LEADZ (1) has the value 31 if BIT_SIZE (1) has the value 32.

16.9.112 LEN (STRING [, KIND])

Description. Length of a character entity.

Class. Inquiry function.

Arguments.

STRING shall be of type character. If it is an unallocated allocatable variable or a pointer that is not associated, its length type parameter shall not be deferred.
KIND (optional) shall be a scalar integer constant expression.

**Result Characteristics.** Integer scalar. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type.

**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 an array.

**Example.** If C is declared by the statement

```fortran
    CHARACTER (11) C (100)
```

LEN (C) has the value 11.

### 16.9.113 LEN_TRIM (STRING [, KIND])

**Description.** Length without trailing blanks.

**Class.** Elemental function.

**Arguments.**
- STRING shall be of type character.
- KIND (optional) shall be a scalar integer constant expression.

**Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type.

**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 (' A B ') has the value 4 and LEN_TRIM (' ') has the value 0.

### 16.9.114 LGE (STRING_A, STRING_B)

**Description.** ASCII greater than or equal.

**Class.** Elemental function.

**Arguments.**
- STRING_A shall be default character or ASCII character.
- STRING_B shall be of type character with the same kind type parameter as STRING_A.

**Result Characteristics.** 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 processor dependent. 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.

**Example.** LGE ('ONE', 'TWO') has the value false.
16.9.115  LGT (STRING_A, STRING_B)

1   Description.  ASCII greater than.
2   Class.  Elemental function.
3   Arguments.
4   STRING_A shall be default character or ASCII character.
5   STRING_B shall be of type character with the same kind type parameter as STRING_A.
6   Result Characteristics.  Default logical.
7   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 processor dependent.  The result is true if STRING_A follows STRING_B in the ASCII collating sequence; otherwise, the result is false.

NOTE 16.19
The result is false if both STRING_A and STRING_B are of zero length.

6   Example.  LGT (‘ONE’, ‘TWO’) has the value false.

16.9.116  LLE (STRING_A, STRING_B)

1   Description.  ASCII less than or equal.
2   Class.  Elemental function.
3   Arguments.
4   STRING_A shall be default character or ASCII character.
5   STRING_B shall be of type character with the same kind type parameter as STRING_A.
6   Result Characteristics.  Default logical.
7   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 processor dependent.  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 16.20
The result is true if both STRING_A and STRING_B are of zero length.

6   Example.  LLE (‘ONE’, ‘TWO’) has the value true.

16.9.117  LLT (STRING_A, STRING_B)

1   Description.  ASCII less than.
2   Class.  Elemental function.
3   Arguments.
4   STRING_A shall be default character or ASCII character.
5   STRING_B shall be of type character with the same kind type parameter as STRING_A.
6   Result Characteristics.  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 processor dependent. The result is true if STRING_A precedes STRING_B in the ASCII collating sequence; otherwise, the result is false.

NOTE 16.21
The result is false if both STRING_A and STRING_B are of zero length.

Example. LLT ("ONE", "TWO") has the value true.

16.9.118 LOG (X)

Description. Natural logarithm.

Class. Elemental function.

Argument. X shall be of type real or complex. If X is real, its value shall be greater than zero. If X is complex, its value shall not be zero.

Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to log_e X. A result of type complex is the principal value with imaginary part ω in the range −π ≤ ω ≤ π. If the real part of X is less than zero and the imaginary part of X is zero, then the imaginary part of the result is approximately π if the imaginary part of X is positive real zero or the processor does not distinguish between positive and negative real zero, and approximately −π if the imaginary part of X is negative real zero.

Example. LOG (10.0) has the value 2.3025851 (approximately).

16.9.119 LOG_GAMMA (X)

Description. Logarithm of the absolute value of the gamma function.

Class. Elemental function.

Argument. X shall be of type real. Its value shall not be a negative integer or zero.

Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to the natural logarithm of the absolute value of the gamma function of X.

Example. LOG_GAMMA (3.0) has the value 0.693 (approximately).

16.9.120 LOG10 (X)

Description. Common logarithm.

Class. Elemental function.

Argument. X shall be of type real. The value of X shall be greater than zero.

Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to log_{10} X.

Example. LOG10 (10.0) has the value 1.0 (approximately).
16.9.121 LOGICAL (L [, KIND])

Description. Conversion between kinds of logical.

Class. Elemental function.

Arguments.

L shall be of type logical.

KIND (optional) shall be a scalar integer constant expression.

Result Characteristics. Logical. If KIND is present, the kind type parameter is that specified by the value of
KIND; otherwise, the kind type parameter is that of default logical.

Result Value. The value is that of L.

Example. LOGICAL (L .OR. .NOT. L) has the value true and is default logical, regardless of the kind type
parameter of the logical variable L.

16.9.122 MASKL (I [, KIND])

Description. Left justified mask.

Class. Elemental function.

Arguments.

I shall be of type integer. It shall be nonnegative and less than or equal to the number of bits z of
the model integer defined for bit manipulation contexts in 16.3 for the kind of the result.

KIND (optional) shall be a scalar integer constant expression.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of
KIND; otherwise, the kind type parameter is that of default integer type.

Result Value. The result value has its leftmost I bits set to 1 and the remaining bits set to 0. The model for
the interpretation of an integer value as a sequence of bits is in 16.3.

Example. MASKL (3) has the value SHIFTL (7, BIT_SIZE (0) − 3).

16.9.123 MASKR (I [, KIND])

Description. Right justified mask.

Class. Elemental function.

Arguments.

I shall be of type integer. It shall be nonnegative and less than or equal to the number of bits z of
the model integer defined for bit manipulation contexts in 16.3 for the kind of the result.

KIND (optional) shall be a scalar integer constant expression.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of
KIND; otherwise, the kind type parameter is that of default integer type.

Result Value. The result value has its rightmost I bits set to 1 and the remaining bits set to 0. The model for
the interpretation of an integer value as a sequence of bits is in 16.3.

Example. MASKR (3) has the value 7.
16.9.124 MATMUL (MATRIX_A, MATRIX_B)

Description. Matrix multiplication.

Class. Transformational function.

Arguments. 
- MATRIX_A shall be a rank-one or rank-two array of numeric type or logical type.
- MATRIX_B shall be of numeric type if MATRIX_A is of numeric type and of logical type if MATRIX_A is of logical type. It shall be an array of rank one or two. MATRIX_A and MATRIX_B shall not both have rank one. The size of the first (or only) dimension of MATRIX_B shall equal the size of the last (or only) dimension of MATRIX_A.

Result Characteristics. If the arguments are of numeric type, the type and kind type parameter of the result are determined by the types of the arguments as specified in 10.1.9.3 for the * operator. If the arguments are of type logical, the result is of type logical with the kind type parameter of the arguments as specified in 10.1.9.3 for the .AND. operator. The shape of the result depends on the shapes of the arguments as follows:

Case (i): If MATRIX_A has shape \([n, m]\) and MATRIX_B has shape \([m, k]\), the result has shape \([n, k]\).
Case (ii): If MATRIX_A has shape \([m]\) and MATRIX_B has shape \([m, k]\), the result has shape \([k]\).
Case (iii): If MATRIX_A has shape \([n, m]\) and MATRIX_B has shape \([m]\), the result has shape \([n]\).

Result Value. 

Case (i): Element \((i, j)\) of the result has the value \(\text{SUM}(\text{MATRIX}_A (i, :) \times \text{MATRIX}_B (:, j))\) if the arguments are of numeric type and has the value \(\text{ANY}(\text{MATRIX}_A (i, :) \text{AND} \text{MATRIX}_B (:, j))\) if the arguments are of logical type.
Case (ii): Element \((j)\) of the result has the value \(\text{SUM}(\text{MATRIX}_A (:) \times \text{MATRIX}_B (:, j))\) if the arguments are of numeric type and has the value \(\text{ANY}(\text{MATRIX}_A (:) \text{AND} \text{MATRIX}_B (:, j))\) if the arguments are of logical type.
Case (iii): Element \((i)\) of the result has the value \(\text{SUM}(\text{MATRIX}_A (i, :) \times \text{MATRIX}_B (:))\) if the arguments are of numeric type and has the value \(\text{ANY}(\text{MATRIX}_A (i, :) \text{AND} \text{MATRIX}_B (:))\) if the arguments are of logical type.

Examples. Let A and B be the matrices

\[
\begin{bmatrix}
1 & 2 & 3 \\
2 & 3 & 4
\end{bmatrix}
\]

and

\[
\begin{bmatrix}
1 & 2 \\
2 & 3 \\
3 & 4
\end{bmatrix}
\]: let X and Y be the vectors [1, 2] and [1, 2, 3].

Case (i): The result of MATMUL (A, B) is the matrix-matrix product AB with the value

\[
\begin{bmatrix}
14 & 20 \\
20 & 29
\end{bmatrix}
\].

Case (ii): The result of MATMUL (X, A) is the vector-matrix product XA with the value [5, 8, 11].
Case (iii): The result of MATMUL (A, Y) is the vector-matrix product AY with the value [14, 20].

16.9.125 MAX (A1, A2 [, A3, ...])

Description. Maximum value.

Class. Elemental function.

Arguments. The arguments shall all have the same type which shall be integer, real, or character and they shall all have the same kind type parameter.

Result Characteristics. The type and kind type parameter of the result are the same as those of the arguments. For arguments of character type, the length of the result is the length of the longest argument.

Result Value. The value of the result is that of the largest argument. For arguments of character type, the result is the value that would be selected by application of intrinsic relational operators; that is, the collating
sequence for characters with the kind type parameter of the arguments is applied. If the selected argument is shorter than the longest argument, the result is extended with blanks on the right to the length of the longest argument.

Examples. MAX (−9.0, 7.0, 2.0) has the value 7.0, MAX ('Z', 'BB') has the value 'Z ', and MAX (["A", 'Z'], ['BB', 'Y ']) has the value ['BB', 'Z '].

16.9.126 MAXEXPONENT (X)

1 Description. Maximum exponent of a real model.

2 Class. Inquiry function.

3 Argument. X shall be of type real. It may be a scalar or an array.

4 Result Characteristics. Default integer scalar.

5 Result Value. The result has the value $e_{\text{max}}$, as defined in 16.4 for the model representing numbers of the same type and kind type parameter as X.

6 Example. MAXEXPONENT (X) has the value 127 for real X whose model is as in Note 16.4.

16.9.127 MAXLOC (ARRAY, DIM [, MASK, KIND, BACK]) or MAXLOC (ARRAY [, MASK, KIND, BACK])

1 Description. Location(s) of maximum value.

2 Class. Transformational function.

3 Arguments.

ARRAY shall be an array of type integer, real, or character.

DIM shall be an integer scalar with a value in the range $1 \leq \text{DIM} \leq n$, where $n$ is the rank of ARRAY.

MASK (optional) shall be of type logical and shall be conformable with ARRAY.

KIND (optional) shall be a scalar integer constant expression.

BACK (optional) shall be a logical scalar.

4 Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type. If DIM does not appear, the result is an array of rank one and of size equal to the rank of ARRAY; otherwise, the result is of rank $n - 1$ and shape $[d_1, d_2, \ldots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \ldots, d_n]$, where $[d_1, d_2, \ldots, d_n]$ is the shape of ARRAY.

5 Result Value.

Case (i): If DIM does not appear and MASK is absent, the result is a rank-one array whose element values are the values of the subscripts of an element of ARRAY whose value equals the maximum value of all of the elements of ARRAY. The $i^{th}$ subscript returned lies in the range 1 to $e_i$, where $e_i$ is the extent of the $i^{th}$ dimension of ARRAY. If ARRAY has size zero, all elements of the result are zero.

Case (ii): If DIM does not appear and MASK is present, the result is a rank-one array whose element values are the values of the subscripts of an element of ARRAY, corresponding to a true element of MASK, whose value equals the maximum value of all such elements of ARRAY. The $i^{th}$ subscript returned lies in the range 1 to $e_i$, where $e_i$ is the extent of the $i^{th}$ dimension of ARRAY. If ARRAY has size zero or every element of MASK has the value false, all elements of the result are zero.

Case (iii): If ARRAY has rank one and DIM is specified, the result has a value equal to that of the first element of MAXLOC (ARRAY [, MASK = MASK, KIND = KIND, BACK = BACK]). Otherwise, if DIM is specified, the value of element $(s_1, s_2, \ldots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \ldots, s_n)$ of the result is equal to
MAXLOC (ARRAY (s₁, s₂, ..., s_{DIM−1}, :, s_{DIM+1}, ..., sₙ),
DIM = 1
[, MASK = MASK (s₁, s₂, ..., s_{DIM−1}, :, s_{DIM+1}, ..., sₙ),
KIND = KIND,
BACK = BACK]).

If only one element has the maximum value, that element’s subscripts are returned. Otherwise, if more than one element has the maximum value and BACK is absent or present with the value false, the element whose subscripts are returned is the first such element, taken in array element order. If BACK is present with the value true, the element whose subscripts are returned is the last such element, taken in array element order.

If ARRAY has type character, the result is the value that would be selected by application of intrinsic relational operators; that is, the collating sequence for characters with the kind type parameter of the arguments is applied.

Examples.

Case (i): The value of MAXLOC ([2, 6, 4, 6]) is [2] and the value of MAXLOC ([2, 6, 4, 6], BACK=.TRUE.) is [4].

Case (ii): If A has the value

\[
\begin{bmatrix}
0 & -5 & 8 & -3 \\
3 & 4 & -1 & 2 \\
1 & 5 & 6 & -4
\end{bmatrix}
\]

MAXLOC (A, MASK = A < 6) has the value [3, 2]. This is independent of the declared lower bounds for A.

Case (iii): The value of MAXLOC ([5, −9, 3], DIM = 1) is 1. If B has the value

\[
\begin{bmatrix}
1 & 3 & -9 \\
2 & 2 & 6
\end{bmatrix}
\]

MAXLOC (B, DIM = 1) is [2, 1, 2] and MAXLOC (B, DIM = 2) is [2, 3]. This is independent of the declared lower bounds for B.

16.9.128 MAXVAL (ARRAY, DIM [, MASK]) or MAXVAL (ARRAY [, MASK])

Description. Maximum value(s) of array.

Class. Transformational function.

Arguments.

ARRAY shall be an array of type integer, real, or character.
DIM shall be an integer scalar with a value in the range 1 ≤ DIM ≤ n, where n is the rank of ARRAY.
MASK (optional) shall be of type logical and shall be conformable with ARRAY.

Result Characteristics. The result is of the same type and type parameters as ARRAY. It is scalar if DIM does not appear; otherwise, the result has rank n − 1 and shape [d₁, d₂, ..., d_{DIM−1}, d_{DIM+1}, ..., dₙ] where [d₁, d₂, ..., dₙ] is the shape of ARRAY.

Result Value.

Case (i): The result of MAXVAL (ARRAY) has a value equal to the maximum value of all the elements of ARRAY if the size of ARRAY is not zero. If ARRAY has size zero and type integer or real, the result has the value of the negative number of the largest magnitude supported by the processor for numbers of the type and kind type parameter of ARRAY. If ARRAY has size zero and type character, the result has the value of a string of characters of length LEN (ARRAY), with each character equal to CHAR (0, KIND (ARRAY)).

Case (ii): The result of MAXVAL (ARRAY, MASK = MASK) has a value equal to that of MAXVAL (PACK (ARRAY, MASK)).

Case (iii): The result of MAXVAL (ARRAY, DIM = DIM [,MASK = MASK]) has a value equal to that of MAXVAL (ARRAY [,MASK = MASK]) if ARRAY has rank one. Otherwise, the value of element (s₁, s₂, ..., s_{DIM−1}, s_{DIM+1}, ..., sₙ) of the result is equal to

\[
\text{MAXVAL (ARRAY (s₁, s₂, ..., s_{DIM−1}, :, s_{DIM+1}, ..., sₙ) [, MASK = MASK (s₁, s₂, ..., s_{DIM−1}, :, s_{DIM+1}, ..., sₙ)])}
\]
If ARRAY is of type character, the result is the value that would be selected by application of intrinsic relational operators; that is, the collating sequence for characters with the kind type parameter of the arguments is applied.

Examples.

Case (i): The value of MAXVAL (\([1, 2, 3]\)) is 3.

Case (ii): MAXVAL \((C, \text{MASK} = C < 0.0)\) is the maximum of the negative elements of \(C\).

Case (iii): If \(B\) is the array \(\begin{bmatrix} 1 & 3 & 5 \\ 2 & 7 & 6 \end{bmatrix}\), \(\text{MAXVAL (B, DIM=1)}\) is \([2, 7, 6]\) and \(\text{MAXVAL (B, DIM=2)}\) is \([5, 7]\).

16.9.129 MERGE (TSOURCE, FSOURCE, MASK)

Description. Expression value selection.

Class. Elemental function.

Arguments.

TSOURCE may be of any type.

FSOURCE shall be of the same type and type parameters as TSOURCE.

MASK shall be of type logical.

Result Characteristics. Same type and type parameters as TSOURCE. Because TSOURCE and FSOURCE are required to have the same type and type parameters (for both the declared and dynamic types), the result is polymorphic if and only if both TSOURCE and FSOURCE are polymorphic.

Result Value. The result is TSOURCE if MASK is true and FSOURCE otherwise.

Examples. If TSOURCE is the array \(\begin{bmatrix} 1 & 6 & 5 \\ 2 & 4 & 6 \end{bmatrix}\), FSOURCE is the array \(\begin{bmatrix} 0 & 3 & 2 \\ 7 & 4 & 8 \end{bmatrix}\) and MASK is the array \(\begin{bmatrix} T & T \\ . & . \end{bmatrix}\), where “T” represents true and “.” represents false, then MERGE (TSOURCE, FSOURCE, MASK) is \(\begin{bmatrix} 1 & 3 & 5 \\ 7 & 4 & 6 \end{bmatrix}\). The value of MERGE (1.0, 0.0, \(K > 0\)) is 1.0 for \(K = 5\) and 0.0 for \(K = -2\).

16.9.130 MERGE_BITS (I, J, MASK)

Description. Merge of bits under mask.

Class. Elemental function.

Arguments.

I shall be of type integer or a \textit{boz-literal-constant}.

J shall be of type integer or a \textit{boz-literal-constant}. If both I and J are of type integer they shall have the same kind type parameter. I and J shall not both be \textit{boz-literal-constants}.

MASK shall be of type integer or a \textit{boz-literal-constant}. If MASK is of type integer, it shall have the same kind type parameter as each other argument of type integer.

Result Characteristics. Same as I if I is of type integer; otherwise, same as J.

Result Value. If any argument is a \textit{boz-literal-constant}, it is first converted as if by the intrinsic function INT to the type and kind type parameter of the result. The result has the value of IOR (IAND (I, MASK), IAND (J, NOT (MASK))).

Example. MERGE_BITS (13, 18, 22) has the value 4.
16.9.131  MIN (A1, A2 [, A3, ...])

1 Description. Minimum value.

2 Class. Elemental function.

3 Arguments. The arguments shall all be of the same type which shall be integer, real, or character and they shall all have the same kind type parameter.

4 Result Characteristics. The type and kind type parameter of the result are the same as those of the arguments. For arguments of character type, the length of the result is the length of the longest argument.

5 Result Value. The value of the result is that of the smallest argument. For arguments of character type, the result is the value that would be selected by application of intrinsic relational operators; that is, the collating sequence for characters with the kind type parameter of the arguments is applied. If the selected argument is shorter than the longest argument, the result is extended with blanks on the right to the length of the longest argument.

6 Examples. MIN (−9.0, 7.0, 2.0) has the value −9.0, MIN ('A', 'YY') has the value 'A ', and MIN (["Z", 'A'], ['YY', 'B ']) has the value ['YY', 'A '].

16.9.132  MINEXPONENT (X)

1 Description. Minimum exponent of a real model.

2 Class. Inquiry function.

3 Argument. X shall be of type real. It may be a scalar or an array.

4 Result Characteristics. Default integer scalar.

5 Result Value. The result has the value $e_{\text{min}}$, as defined in 16.4 for the model representing numbers of the same type and kind type parameter as X.

6 Example. MINEXPONENT (X) has the value −126 for real X whose model is as in Note 16.4.

16.9.133  MINLOC (ARRAY, DIM [, MASK, KIND, BACK]) or MINLOC (ARRAY [, MASK, KIND, BACK])

1 Description. Location(s) of minimum value.

2 Class. Transformational function.

3 Arguments.

4 Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type. If DIM does not appear, the result is an array of rank one and of size equal to the rank of ARRAY; otherwise, the result is of rank $n − 1$ and shape $[d_1, d_2, \ldots, d_{\text{DIM}−1}, d_{\text{DIM}+1}, \ldots, d_n]$, where $[d_1, d_2, \ldots, d_n]$ is the shape of ARRAY.

5 Result Value. Case (i): If DIM does not appear and MASK is absent the result is a rank-one array whose element values are the values of the subscripts of an element of ARRAY whose value equals the minimum value.
of all the elements of ARRAY. The \(i^{th}\) subscript returned lies in the range 1 to \(e_i\), where \(e_i\) is the extent of the \(i^{th}\) dimension of ARRAY. If ARRAY has size zero, all elements of the result are zero.

**Case (ii):** If DIM does not appear and MASK is present, the result is a rank-one array whose element values are the values of the subscripts of an element of ARRAY, corresponding to a true element of MASK, whose value equals the minimum value of all such elements of ARRAY. The \(i^{th}\) subscript returned lies in the range 1 to \(e_i\), where \(e_i\) is the extent of the \(i^{th}\) dimension of ARRAY. If ARRAY has size zero or every element of MASK has the value false, all elements of the result are zero.

**Case (iii):** If ARRAY has rank one and DIM is specified, the result has a value equal to that of the first element of MINLOC (ARRAY [, MASK = MASK, KIND = KIND, BACK = BACK]). Otherwise, if DIM is specified, the value of element \((s_1, s_2, \ldots, s_{DIM-1}, \tilde{s}_{DIM+1}, \ldots, s_n)\) of the result is equal to

\[
\text{MINLOC (ARRAY (} s_1, s_2, \ldots, s_{DIM-1}, \tilde{s}_{DIM+1}, \ldots, s_n),
\text{,) }\]

with

\[
\text{DIM = 1, MASK = MASK (} s_1, s_2, \ldots, s_{DIM-1}, \tilde{s}_{DIM+1}, \ldots, s_n),
\text{,) }\]

\[
\text{KIND = KIND, BACK = BACK).}
\]

If only one element has the minimum value, that element’s subscripts are returned. Otherwise, if more than one element has the minimum value and BACK is absent or present with the value false, the element whose subscripts are returned is the first such element, taken in array element order. If BACK is present with the value true, the element whose subscripts are returned is the last such element, taken in array element order.

If ARRAY is of type character, the result is the value that would be selected by application of intrinsic relational operators; that is, the collating sequence for characters with the kind type parameter of the arguments is applied.

**Examples.**

**Case (i):** The value of MINLOC ([4, 3, 6, 3]) is [2] and the value of MINLOC ([4, 3, 6, 3], BACK = .TRUE.) is [4].

**Case (ii):** If A has the value

\[
\begin{pmatrix}
0 & -5 & 8 & -3 \\
3 & 4 & -1 & 2 \\
1 & 5 & 6 & -4
\end{pmatrix}
\]

MINLOC (A, MASK = A > -4) has the value [1, 4]. This is independent of the declared lower bounds for A.

**Case (iii):** The value of MINLOC ([5, -9, 3], DIM = 1) is 2. If B has the value

\[
\begin{pmatrix}
1 & 3 & -9 \\
2 & 2 & 6
\end{pmatrix}
\]

MINLOC (B, DIM = 1) is [1, 2, 1] and MINLOC (B, DIM = 2) is [3, 1]. This is independent of the declared lower bounds for B.

### 16.9.134 MINVAL (ARRAY, DIM [, MASK]) or MINVAL (ARRAY [, MASK])

**Description.** Minimum value(s) of array.

**Class.** Transformational function.

**Arguments.**

- ARRAY shall be an array of type integer, real, or character.
- DIM shall be an integer scalar with a value in the range 1 ≤ DIM ≤ n, where n is the rank of ARRAY.
- MASK (optional) shall be of type logical and shall be conformable with ARRAY.

**Result Characteristics.** The result is of the same type and type parameters as ARRAY. It is scalar if DIM does not appear; otherwise, the result has rank \(n - 1\) and shape \([d_1, d_2, \ldots, d_{DIM-1}, d_{DIM+1}, \ldots, d_n]\) where \([d_1, d_2, \ldots, d_n]\) is the shape of ARRAY.

**Result Value.**

**Case (i):** The result of MINVAL (ARRAY) has a value equal to the minimum value of all the elements of ARRAY if the size of ARRAY is not zero. If ARRAY has size zero and type integer or real, the result has the value of the positive number of the largest magnitude supported by the processor for numbers of the type and kind type parameter of ARRAY. If ARRAY has size zero and type
character, the result has the value of a string of characters of length \( \text{LEN} (\text{ARRAY}) \), with each character equal to \( \text{CHAR} (n \to \text{KIND} (\text{ARRAY})) \), where \( n \) is the number of characters in the collating sequence for characters with the kind type parameter of \( \text{ARRAY} \).

**Case (ii):** The result of \( \text{MINVAL} (\text{ARRAY}, \text{MASK} = \text{MASK}) \) has a value equal to that of \( \text{MINVAL} (\text{PACK} (\text{ARRAY}, \text{MASK})) \).

**Case (iii):** The result of \( \text{MINVAL} (\text{ARRAY}, \text{DIM} = \text{DIM} [= \text{MASK} = \text{MASK}]) \) has a value equal to that of \( \text{MINVAL} (\text{ARRAY} [= \text{MASK} = \text{MASK}]) \) if \( \text{ARRAY} \) has rank one. Otherwise, the value of element \( (s_1, s_2, \ldots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \ldots, s_n) \) of the result is equal to

\[
\text{MINVAL} (\text{ARRAY} (s_1, s_2, \ldots, s_{\text{DIM}-1}, \vdots, s_{\text{DIM}+1}, \ldots, s_n) [, \text{MASK} = \text{MASK} (s_1, s_2, \ldots, s_{\text{DIM}-1}, \vdots, s_{\text{DIM}+1}, \ldots, s_n) ]).
\]

If \( \text{ARRAY} \) is of type character, the result is the value that would be selected by application of intrinsic relational operators; that is, the collating sequence for characters with the kind type parameter of the arguments is applied.

**Examples.**

**Case (i):** The value of \( \text{MINVAL} ([1, 2, 3]) \) is 1.

**Case (ii):** \( \text{MINVAL} (C, \text{MASK} = C \gt 0.0) \) is the minimum of the positive elements of \( C \).

**Case (iii):** If \( B \) is the array

\[
\begin{bmatrix}
1 & 3 & 5 \\
2 & 4 & 6
\end{bmatrix}
\]

\( \text{MINVAL} (B, \text{DIM} = 1) \) is \([1, 3, 5]\) and \( \text{MINVAL} (B, \text{DIM} = 2) \) is \([1, 2]\).

### 16.9.135 MOD (A, P)

**Description.** Remainder function.

**Class.** Elemental function.

**Arguments.**

- \( A \) shall be of type integer or real.
- \( P \) shall be of the same type and kind type parameter as \( A \). \( P \) shall not be zero.

**Result Characteristics.** Same as \( A \).

**Result Value.** The value of the result is \( A - \text{INT}(A/P) \times P \).

**Examples.** \( \text{MOD} (3.0, 2.0) \) has the value 1.0 (approximately). \( \text{MOD} (8, 5) \) has the value 3. \( \text{MOD} (-8, 5) \) has the value \(-3\). \( \text{MOD} (8, -5) \) has the value 3. \( \text{MOD} (-8, -5) \) has the value \(-3\).

### 16.9.136 MODULO (A, P)

**Description.** Modulo function.

**Class.** Elemental function.

**Arguments.**

- \( A \) shall be of type integer or real.
- \( P \) shall be of the same type and kind type parameter as \( A \). \( P \) shall not be zero.

**Result Characteristics.** Same as \( A \).

**Result Value.**

**Case (i):** \( A \) is of type integer. \( \text{MODULO} (A, P) \) has the value \( R \) such that \( A = Q \times P + R \), where \( Q \) is an integer, the inequalities \( 0 \leq R < P \) hold if \( P > 0 \), and \( P < R \leq 0 \) hold if \( P < 0 \).

**Case (ii):** \( A \) is of type real. The value of the result is \( A - \text{FLOOR}(A/P) \times P \).
**Examples.** MODULO (8, 5) has the value 3. MODULO (−8, 5) has the value 2. MODULO (8, −5) has the value −3.

**16.9.137 MOVE_ALLOC (FROM, TO [, STAT, ERRMSG])**

- **Description.** Move an allocation.
- **Class.** Pure subroutine.

**Arguments.**
- FROM may be of any type, rank, and corank. It shall be allocatable and shall not be a coindexed object. It is an INTENT (INOUT) argument.
- TO shall be type compatible (7.3.2.3) with FROM and have the same rank and corank. It shall be allocatable and shall not be a coindexed object. It shall be polymorphic if FROM is polymorphic. It is an INTENT (OUT) argument. Each nondeferred parameter of the declared type of TO shall have the same value as the corresponding parameter of the declared type of FROM.
- STAT (optional) shall be a noncoindexed default integer scalar. It is an INTENT (OUT) argument.
- ERRMSG (optional) shall be a noncoindexed default character scalar. It is an INTENT (INOUT) argument.

If execution of MOVE_ALLOC is successful, or if STAT_FAILED_IMAGE is assigned to STAT,

- On invocation of MOVE_ALLOC, if the allocation status of TO is allocated, it is deallocated. Then, if FROM has an allocation status of allocated on entry to MOVE_ALLOC, TO becomes allocated with dynamic type, type parameters, bounds, cobounds, and value identical to those that FROM had on entry to MOVE_ALLOC. Note that if FROM and TO are the same variable, it shall be unallocated when MOVE_ALLOC is invoked.
- If TO has the TARGET attribute, any pointer associated with FROM on entry to MOVE_ALLOC becomes correspondingly associated with TO. If TO does not have the TARGET attribute, the pointer association status of any pointer associated with FROM on entry becomes undefined.
- The allocation status of FROM becomes unallocated.

When a reference to MOVE_ALLOC is executed for which the FROM argument is a coarray, there is an implicit synchronization of all active images of the current team. On those images, execution of the segment (11.6.2) following the CALL statement is delayed until all other active images of the current team have executed the same statement the same number of times.

If STAT is present and execution is successful, it is assigned the value zero.

If an error condition occurs and STAT is absent, error termination is initiated. Otherwise, if the current team contains a stopped image, STAT is assigned the value STAT_STOPPED_IMAGE from the intrinsic module ISO_FORTRAN_ENV. Otherwise, if the current team contains a failed image, no stopped image, and no other error condition occurs, STAT is assigned the value STAT_FAILED_IMAGE from the intrinsic module ISO_FORTRAN_ENV. Otherwise, STAT is assigned a processor-dependent positive value that differs from that of STAT_STOPPED_IMAGE or STAT_FAILED_IMAGE.

If the ERRMSG argument is present and an error condition occurs, it is assigned an explanatory message. If no error condition occurs, the definition status and value of ERRMSG are unchanged.

**Example.** The example below demonstrates reallocation of GRID to twice its previous size, with its previous contents evenly distributed over the new elements so that intermediate points can be inserted.

```fortran
REAL,ALLOCATABLE :: GRID(:),TEMPGRID(:)
...
ALLOCATE(GRID(-N:N)) ! initial allocation of GRID
...```
ALLOCATE(TEMPGRID(-2*N:2*N)) ! allocate bigger grid
TEMPPGRID(::2)=GRID ! distribute values to new locations
CALL MOVE_ALLOC(TO=GRID,FROM=TEMPPGRID)

The old grid is deallocated because TO is INTENT (OUT), and GRID then takes over the new grid allocation.

NOTE 16.22
It is expected that the implementation of allocatable objects will typically involve descriptors to locate the allocated storage; MOVE_ALLOC could then be implemented by transferring the contents of the descriptor for FROM to the descriptor for TO and clearing the descriptor for FROM.

16.9.138 MVBITS (FROM, FROMPOS, LEN, TO, TOPOS)

Description. Copy a sequence of bits.

Class. Elemental subroutine.

Arguments.
FROM shall be of type integer. It is an INTENT (IN) argument.
FROMPOS shall be of type integer and nonnegative. It is an INTENT (IN) argument. FROMPOS + LEN shall be less than or equal to BIT_SIZE (FROM). The model for the interpretation of an integer value as a sequence of bits is in 16.3.
LEN shall be of type integer and nonnegative. It is an INTENT (IN) argument.
TO shall be a variable of the same type and kind type parameter value as FROM and may be associated with FROM (15.8.3). It is an INTENT (INOUT) argument. TO is defined 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 model for the interpretation of an integer value as a sequence of bits is in 16.3.
TOPOS shall be of type integer and nonnegative. It is an INTENT (IN) argument. TOPOS + LEN shall be less than or equal to BIT_SIZE (TO).

Example. If TO has the initial value 6, its value after the statement CALL MVBITS (7, 2, 2, TO, 0) is 5.

16.9.139 NEAREST (X, S)

Description. Adjacent machine number.

Class. Elemental function.

Arguments.
X shall be of type real.
S shall be of type real and not equal to zero.

Result Characteristics. Same as X.

Result Value. The result has a value equal to the machine-representable number distinct from X and nearest to it in the direction of the \( \infty \) with the same sign as S.

Example. NEAREST (3.0, 2.0) has the value \( 3 + 2^{-22} \) on a machine whose representation is that of the model in Note 16.4.
NOTE 16.23

Unlike other floating-point manipulation functions, NEAREST operates on machine-representable numbers rather than model numbers. On many systems there are machine-representable numbers that lie between adjacent model numbers.

16.9.140 NEW_LINE (A)

1 Description. Newline character.

2 Class. Inquiry function.

3 Argument. A shall be of type character. It may be a scalar or an array.

4 Result Characteristics. Character scalar of length one with the same kind type parameter as A.

5 Result Value.

  Case (i): If A is default character and the character in position 10 of the ASCII collating sequence is representable in the default character set, then the result is ACHAR (10).

  Case (ii): If A is ASCII character or ISO 10646 character, then the result is CHAR (10, KIND (A)).

  Case (iii): Otherwise, the result is a processor-dependent character that represents a newline in output to files connected for formatted stream output if there is such a character.

  Case (iv): Otherwise, the result is the blank character.

6 Example. If there is a suitable newline character, and unit 10 is connected for formatted stream output, the statement

\[
\text{WRITE (10, '(A)') 'New'//NEW_LINE('a')//'Line'}
\]

will write a record containing “New” and then a record containing “Line”.

16.9.141 NINT (A [, KIND])

1 Description. Nearest integer.

2 Class. Elemental function.

3 Arguments.

A shall be of type real.

KIND (optional) shall be a scalar integer constant expression.

4 Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default integer type.

5 Result Value. The result is the integer nearest A, or if there are two integers equally near A, the result is whichever such integer has the greater magnitude.

6 Example. NINT (2.783) has the value 3.

16.9.142 NORM2 (X) or NORM2 (X, DIM)

1 Description. \( L_2 \) norm of an array.

2 Class. Transformational function.

3 Arguments.

X shall be a real array.
DIM shall be an integer scalar with a value in the range \(1 \leq \text{DIM} \leq n\), where \(n\) is the rank of X.

**Result Characteristics.** The result is of the same type and type parameters as X. It is scalar if DIM does not appear; otherwise the result has rank \(n - 1\) and shape \([d_1, d_2, \ldots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \ldots, d_n]\), where \(n\) is the rank of X and \([d_1, d_2, \ldots, d_n]\) is the shape of X.

**Result Value.**

-CASE (i): The result of NORM2 (X) has a value equal to a processor-dependent approximation to the generalized \(L_2\) norm of X, which is the square root of the sum of the squares of the elements of X. If X has size zero, the result has the value zero.

-CASE (ii): The result of NORM2 (X, DIM=DIM) has a value equal to that of NORM2 (X) if X has rank one. Otherwise, the value of element \((s_1, s_2, \ldots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \ldots, s_n)\) of the result is equal to NORM2 \((X(s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n))\).

It is recommended that the processor compute the result without undue overflow or underflow.

**Example.** The value of NORM2 ([3.0, 4.0]) is 5.0 (approximately). If X has the value \[
\begin{pmatrix}
1.0 & 2.0 \\
3.0 & 4.0
\end{pmatrix}
\] then the value of NORM2 (X, DIM=1) is [3.162, 4.472] (approximately) and the value of NORM2 (X, DIM=2) is [2.236, 5.0] (approximately).

16.9.143 NOT (I)

-**Description.** Bitwise complement.

-**Class.** Elemental function.

-**Argument.** I shall be of type integer.

-**Result Characteristics.** 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>
<th>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 model for the interpretation of an integer value as a sequence of bits is in 16.3.

**Example.** If I is represented by the string of bits 01010101, NOT (I) has the binary value 10101010.

16.9.144 NULL ([MOLD])

-**Description.** Disassociated pointer or unallocated allocatable entity.

-**Class.** Transformational function.

-**Argument.** MOLD shall be a pointer or allocatable. It may be of any type or may be a procedure pointer. If MOLD is a pointer its pointer association status may be undefined, disassociated, or associated. If MOLD is allocatable its allocation status may be allocated or unallocated. It need not be defined with a value.

-**Result Characteristics.** If MOLD is present, the characteristics are the same as MOLD. If MOLD has deferred type parameters, those type parameters of the result are deferred.

If MOLD is absent, the characteristics of the result are determined by the entity with which the reference is associated. See Table 16.5. MOLD shall not be absent in any other context. If any type parameters of the contextual entity are deferred, those type parameters of the result are deferred. If any type parameters of the contextual entity are assumed, MOLD shall be present.
If the context of the reference to NULL is an actual argument in a generic procedure reference, MOLD shall be present if the type, type parameters, or rank are required to resolve the generic reference.

Table 16.5: Characteristics of the result of NULL ( )

<table>
<thead>
<tr>
<th>Appearance of NULL ( )</th>
<th>Type, type parameters, and rank of result:</th>
</tr>
</thead>
<tbody>
<tr>
<td>right side of a pointer assignment</td>
<td>pointer on the left side</td>
</tr>
<tr>
<td>initialization for an object in a declaration</td>
<td>the object</td>
</tr>
<tr>
<td>default initialization for a component</td>
<td>the component</td>
</tr>
<tr>
<td>in a structure constructor</td>
<td>the corresponding component</td>
</tr>
<tr>
<td>as an actual argument</td>
<td>the corresponding dummy argument</td>
</tr>
<tr>
<td>in a DATA statement</td>
<td>the corresponding pointer object</td>
</tr>
</tbody>
</table>

Result. The result is a disassociated pointer or an unallocated allocatable entity.

Examples.

Case (i): REAL, POINTER, DIMENSION (:) :: VEC => NULL ( ) defines the initial association status of VEC to be disassociated.

Case (ii): The MOLD argument is required in the following:

```fortran
INTERFACE GEN

SUBROUTINE S1 (J, PI)
    INTEGER J
    INTEGER, POINTER :: PI
END SUBROUTINE S1

SUBROUTINE S2 (K, PR)
    INTEGER K
    REAL, POINTER :: PR
END SUBROUTINE S2

END INTERFACE

REAL, POINTER :: REAL_PTR
CALL GEN (7, NULL (REAL_PTR) ) ! Invokes S2
```

16.9.145 NUM_IMAGES ( ) or NUM_IMAGES (TEAM) or NUM_IMAGES (TEAM_NUMBER)

Description. Number of images.

Class. Transformational function.

Arguments.

TEAM shall be a scalar of type TEAM_TYPE from the intrinsic module ISO_FORTRAN_ENV, with a value that identifies the current or an ancestor team.

TEAM_NUMBER shall be a positive integer scalar. If the current team is not the initial team, it shall identify a team whose parent is the same as that of the current team.

Result Characteristics. Default integer scalar.

Result Value. The number of images in the specified team, or in the current team if no team is specified.

Example. The following code uses image 1 to read data and broadcast it to other images.

```fortran
REAL :: P[*]
```
IF (THIS_IMAGE()==1) THEN
   READ (6,*) P
   DO I = 2, NUM_IMAGES()
      P[I] = P
   END DO
END IF
SYNC ALL

16.9.146 OUT_OF_RANGE (X, MOLD [, ROUND])

Description. Whether a value cannot be converted safely.

Class. Elemental function.

Arguments.
X shall be of type integer or real.
MOLD shall be an integer or real scalar. If it is a variable, it need not be defined.
ROUND (optional) shall be a logical scalar. ROUND shall be present only if X is of type real and MOLD is of type integer.

Result Characteristics. Default logical scalar.

Result Value.
Case (i): If MOLD is of type integer, and ROUND is absent or present with the value false, the result is true if and only if the value of X is an IEEE infinity or NaN, or if the integer with largest magnitude that lies between zero and X inclusive is not representable by objects with the type and kind of MOLD.
Case (ii): If MOLD is of type integer, and ROUND is present with the value true, the result is true if and only if the value of X is an IEEE infinity or NaN, or if the integer nearest X, or the integer of greater magnitude if two integers are equally near to X, is not representable by objects with the type and kind of MOLD.
Case (iii): Otherwise, the result is true if and only if the value of X is an IEEE infinity or NaN that is not supported by objects of the type and kind of MOLD, or if X is a finite number and the result of rounding the value of X (according to the IEEE rounding mode if appropriate) to the extended model for the kind of MOLD has magnitude larger than that of the largest finite number with the same sign as X that is representable by objects with the type and kind of MOLD.

Examples. If INT8 is the kind value for an 8-bit binary integer type, OUT_OF_RANGE (−128.5, 0_INT8) will have the value false and OUT_OF_RANGE (−128.5, 0_INT8, .TRUE.) will have the value true.

16.9.147 PACK (ARRAY, MASK [, VECTOR])

Description. Array packed into a vector.

Class. Transformational function.

Arguments.
ARRAY shall be an array of any type.
MASK shall be of type logical and shall be conformable with ARRAY.
VECTOR (optional) shall be of the same type and type parameters as ARRAY and shall have rank one. VECTOR shall have at least as many elements as there are true elements in MASK. If MASK is scalar with the value true, VECTOR shall have at least as many elements as there are in ARRAY.
Result Characteristics. The result is an array of rank one with the same type and type parameters as ARRAY. If VECTOR is present, the result size is that of VECTOR; otherwise, the result size is the number $t$ of true elements in MASK unless MASK is scalar with the value true, in which case the result size is the size of ARRAY.

Result Value. Element $i$ of the result is the element of ARRAY that corresponds to the $i^{th}$ true element of MASK, taking elements in array element order, for $i = 1, 2, \ldots, t$. If VECTOR is present and has size $n > t$, element $i$ of the result has the value VECTOR ($i$), for $i = t + 1, \ldots, n$.

Examples. The nonzero elements of an array $M$ with the value
\[
\begin{bmatrix}
0 & 0 & 0 \\
9 & 0 & 0 \\
0 & 0 & 7
\end{bmatrix}
\]
can be “gathered” by the function PACK. The result of PACK ($M$, MASK = $M /= 0$) is [9, 7] and the result of PACK ($M$, $M /= 0$, VECTOR = [2, 4, 6, 8, 10, 12]) is [9, 7, 6, 8, 10, 12].

16.9.148 PARITY (MASK) or PARITY (MASK, DIM)

Description. Array reduced by .NEQV. operation.

Class. Transformational function.

Arguments.

MASK shall be a logical array.

DIM shall be an integer scalar with a value in the range $1 \leq \text{DIM} \leq n$, where $n$ is the rank of MASK.

Result Characteristics. The result is of type logical with the same kind type parameter as MASK. It is scalar if DIM does not appear; otherwise, the result has rank $n - 1$ and shape $[d_1, d_2, \ldots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \ldots, d_n]$ where $[d_1, d_2, \ldots, d_n]$ is the shape of MASK.

Result Value.

Case (i): The result of PARITY (MASK) has the value true if an odd number of the elements of MASK are true, and false otherwise.

Case (ii): If MASK has rank one, PARITY (MASK, DIM) is equal to PARITY (MASK). Otherwise, the value of element $(s_1, s_2, \ldots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \ldots, s_n)$ of PARITY (MASK, DIM) is equal to PARITY (MASK $(s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n)$).

Examples.

Case (i): The value of PARITY ([T, T, T, F]) is true if T has the value true and F has the value false.

Case (ii): If B is the array $\begin{bmatrix} T & T & F \\ T & T & T \end{bmatrix}$, where T has the value true and F has the value false, then PARITY (B, DIM=1) has the value [F, F, T] and PARITY (B, DIM=2) has the value [F, T].

16.9.149 POPCNT (I)

Description. Number of one bits.

Class. Elemental function.

Argument. I shall be of type integer.

Result Characteristics. Default integer.

Result Value. The result value is equal to the number of one bits in the sequence of bits of I. The model for the interpretation of an integer value as a sequence of bits is in 16.3.

Examples. POPCNT ([1, 2, 3, 4, 5, 6]) has the value [1, 1, 1, 1, 2, 2].
16.9.150 POPPAR (I)

Description. Parity expressed as 0 or 1.

Class. Elemental function.

Argument. I shall be of type integer.

Result Characteristics. Default integer.

Result Value. POPPAR (I) has the value 1 if POPCNT (I) is odd, and 0 if POPCNT (I) is even.

Examples. POPPAR ([1, 2, 3, 4, 5, 6]) has the value [1, 1, 0, 1, 0, 0].

16.9.151 PRECISION (X)

Description. Decimal precision of a real model.

Class. Inquiry function.

Argument. X shall be of type real or complex. It may be a scalar or an array.

Result Characteristics. Default integer scalar.

Result Value. The result has the value INT ((p − 1) * LOG10 (b)) + k, where b and p are as defined in 16.4 for the model representing real numbers with the same value for the kind type parameter as X, and where k is 1 if b is an integral power of 10 and 0 otherwise.

Example. PRECISION (X) has the value INT (23 * LOG10 (2.)) = INT (6.92...) = 6 for real X whose model is as in Note 16.4.

16.9.152 PRESENT (A)

Description. Presence of optional argument.

Class. Inquiry function.

Argument. A shall be the name of an optional dummy argument that is accessible in the subprogram in which the PRESENT function reference appears. There are no other requirements on A.

Result Characteristics. Default logical scalar.

Result Value. The result has the value true if A is present (15.5.2.12) and otherwise has the value false.

16.9.153 PRODUCT (ARRAY, DIM [, MASK]) or PRODUCT (ARRAY [, MASK])

Description. Array reduced by multiplication.

Class. Transformational function.

Arguments.

ARRAY shall be an array of numeric type.

DIM shall be an integer scalar with a value in the range 1 ≤ DIM ≤ n, where n is the rank of ARRAY.

MASK (optional) shall be of type logical and shall be conformable with ARRAY.

Result Characteristics. The result is of the same type and kind type parameter as ARRAY. It is scalar if DIM does not appear; otherwise, the result has rank n − 1 and shape [d1, d2, ..., dDIM−1, dDIM+1, ..., dn] where [d1, d2, ..., dn] is the shape of ARRAY.
5 Result Value.

Case (i): The result of PRODUCT (ARRAY) has a value equal to a processor-dependent approximation to
the product of all the elements of ARRAY or has the value one if ARRAY has size zero.

Case (ii): The result of PRODUCT (ARRAY, MASK = MASK) has a value equal to a processor-dependent
approximation to the product of the elements of ARRAY corresponding to the true elements of
MASK or has the value one if there are no true elements.

Case (iii): If ARRAY has rank one, PRODUCT (ARRAY, DIM = DIM [, MASK = MASK]) has a value equal
to that of PRODUCT (ARRAY [, MASK = MASK ]). Otherwise, the value of element (s₁, s₂, ..., $s_{DIM-1}$, $s_{DIM+1}$, ..., $s_n$) of PRODUCT (ARRAY, DIM = DIM [, MASK = MASK]) is equal to
PRODUCT (ARRAY (s₁, s₂, ..., $s_{DIM-1}$, :, $s_{DIM+1}$, ..., $s_n$ ) [, MASK = MASK (s₁, s₂, ...,
$s_{DIM-1}$, :, $s_{DIM+1}$, ..., $s_n$ ) ]).

6 Examples.

Case (i): The value of PRODUCT ([1, 2, 3]) is 6.

Case (ii): PRODUCT (C, MASK = C > 0.0) forms the product of the positive elements of C.

Case (iii): If B is the array
\[
\begin{bmatrix}
1 & 3 & 5 \\
2 & 4 & 6
\end{bmatrix}
\]
PRODUCT (B, DIM = 1) is [2, 12, 30] and PRODUCT (B, DIM = 2) is [15, 48].

16.9.154 RADIX (X)

1 Description. Base of a numeric model.

2 Class. Inquiry function.

3 Argument. X shall be of type integer or real. It may be a scalar or an array.

4 Result Characteristics. Default integer scalar.

5 Result Value. The result has the value $r$ if X is of type integer and the value $b$ if X is of type real, where $r$ and $b$ are as defined in 16.4 for the model representing numbers of the same type and kind type parameter as X.

6 Example. RADIX (X) has the value 2 for real X whose model is as in Note 16.4.

16.9.155 RANDOM_INIT (REPEATABLE, IMAGE_DISTINCT)

1 Description. Initialize the pseudorandom number generator.

2 Class. Subroutine.

3 Arguments.

REPEATABLE shall be a logical scalar. It is an INTENT (IN) argument. If it has the value true, the seed
accessed by the pseudorandom number generator is set to a processor-dependent value that is the
same each time RANDOM_INIT is called from the same image. If it has the value false, the seed
is set to a processor-dependent, unpredictably different value on each call.

IMAGE_DISTINCT shall be a logical scalar. It is an INTENT (IN) argument. If it has the value true, the
seed accessed by the pseudorandom number generator is set to a processor-dependent value that
is distinct from the value that would be set by a call to RANDOM_INIT by another image. If
it has the value false, the value to which the seed is set does not depend on which image calls
RANDOM_INIT.

4 Example. The following statement initializes the pseudorandom number generator so that the seed is different
on each call and that the sequence generated will differ from that of another image:

CALL RANDOM_INIT (REPEATABLE=.FALSE., IMAGE_DISTINCT=.TRUE.)
**16.9.156 RANDOM_NUMBER (HARVEST)**

1. **Description.** Generate pseudorandom number(s).

2. **Class.** Subroutine.

3. **Argument.** HARVEST shall be of type real. It is an INTENT (OUT) argument. It may be a scalar or an array. It is assigned pseudorandom numbers from the uniform distribution in the interval $0 \leq x < 1$. If images use a common generator, the interleaving of values assigned in unordered segments is processor dependent.

4. **Example.**

   ```fortran
   REAL X, Y (10, 10)
   ! Initialize X with a pseudorandom number
   CALL RANDOM_NUMBER (HARVEST = X)
   CALL RANDOM_NUMBER (Y)
   ! X and Y contain uniformly distributed random numbers
   ```

**16.9.157 RANDOM_SEED ([SIZE, PUT, GET])**

1. **Description.** Restart or query the pseudorandom number generator.

2. **Class.** Subroutine.

3. **Arguments.** There shall either be exactly one or no arguments present.

   - SIZE (optional) shall be a default integer scalar. It is an INTENT (OUT) argument. It is assigned the number $N$ of integers that the processor uses to hold the value of the seed.

   - PUT (optional) shall be a default integer array of rank one and size $\geq N$. It is an INTENT (IN) argument. It is used in a processor-dependent manner to compute the seed value accessed by the pseudorandom number generator.

   - GET (optional) shall be a default integer array of rank one and size $\geq N$. It is an INTENT (OUT) argument. It is assigned the value of the seed.

4. If no argument is present, the processor assigns a processor-dependent value to the seed.

5. The pseudorandom number generator used by RANDOM_NUMBER maintains a seed that is updated during the execution of RANDOM_NUMBER and that can be retrieved or changed by RANDOM_SEED. Computation of the seed from the argument PUT is performed in a processor-dependent manner. The value assigned to GET need not be the same as the value of PUT in an immediately preceding reference to RANDOM_SEED. For example, following execution of the statements

   ```fortran
   CALL RANDOM_SEED (PUT=SEED1)
   CALL RANDOM_SEED (GET=SEED2)
   ```

   SEED2 need not equal SEED1. When the values differ, the use of either value as the PUT argument in a subsequent call to RANDOM_SEED shall result in the same sequence of pseudorandom numbers being generated. For example, after execution of the statements

   ```fortran
   CALL RANDOM_SEED (PUT=SEED1)
   CALL RANDOM_SEED (GET=SEED2)
   CALL RANDOM_NUMBER (X1)
   CALL RANDOM_SEED (PUT=SEED2)
   CALL RANDOM_NUMBER (X2)
   ```

   X2 equals X1.
Examples.

```fortran
CALL RANDOM_SEED ! Processor initialization
CALL RANDOM_SEED (SIZE = K) ! Puts size of seed in K
CALL RANDOM_SEED (PUT = SEED (1 : K)) ! Define seed
CALL RANDOM_SEED (GET = OLD (1 : K)) ! Read current seed
```

16.9.158 RANGE (X)

1 Description. Decimal exponent range of a numeric model (16.4).
2 Class. Inquiry function.
3 Argument. X shall be of type integer, real, or complex. It may be a scalar or an array.
4 Result Characteristics. Default integer scalar.

5 Result Value.

   Case (i): If X is of type integer, the result has the value INT (LOG10 (HUGE (X))).
   Case (ii): If X is of type real, the result has the value INT (MIN (LOG10 (HUGE (X)), −LOG10 (TINY (X))))).
   Case (iii): If X is of type complex, the result has the value RANGE (REAL (X)).

6 Examples. RANGE (X) has the value 38 for real X whose model is as in Note 16.4, because in this case
HUGE (X) = (1 − 2−24) × 2127 and TINY (X) = 2−127.

16.9.159 RANK (A)

1 Description. Rank of a data object.
2 Class. Inquiry function.
3 Argument. A shall be a data object of any type.
4 Result Characteristics. Default integer scalar.

5 Result Value. The value of the result is the rank of A.
6 Example. If X is an assumed-rank dummy argument and its associated effective argument is an array of rank
3, RANK(X) has the value 3.

16.9.160 REAL (A [, KIND])

1 Description. Conversion to real type.
2 Class. Elemental function.
3 Arguments.
   A shall be of type integer, real, or complex, or a `boz-literal-constant`.
   KIND (optional) shall be a scalar integer constant expression.
4 Result Characteristics. Real.

   Case (i): If A is of type integer or real and KIND is present, the kind type parameter is that specified by the
   value of KIND. If A is of type integer or real and KIND is not present, the kind type parameter is
   that of default real kind.
   Case (ii): If A is of type complex and KIND is present, the kind type parameter is that specified by the value
   of KIND. If A is of type complex and KIND is not present, the kind type parameter is the kind
   type parameter of A.
Case (iii): If A is a \textit{boz-literal-constant} and KIND is present, the kind type parameter is that specified by the value of KIND. If A is a \textit{boz-literal-constant} and KIND is not present, the kind type parameter is that of default real kind.

5 Result Value.

Case (i): If A is of type integer or real, the result is equal to a processor-dependent approximation to A.

Case (ii): If A is of type complex, the result is equal to a processor-dependent approximation to the real part of A.

Case (iii): If A is a \textit{boz-literal-constant}, the value of the result is the value whose internal representation as a bit sequence is the same as that of A as modified by padding or truncation according to 16.3.3. The interpretation of the bit sequence is processor dependent.

6 Examples. REAL (−3) has the value −3.0. REAL (Z) has the same kind type parameter and the same value as the real part of the complex variable Z.

16.9.161 REDUCE (ARRAY, OPERATION [], MASK, IDENTITY, ORDERED]) or REDUCE (ARRAY, OPERATION, DIM [], MASK, IDENTITY, ORDERED])

1 Description. General reduction of array.

2 Class. Transformational function.

3 Arguments.

ARRAY shall be an array of any type.

OPERATION shall be a \textit{pure function} with two arguments of the same type and \textit{type parameters} as ARRAY. Its result shall have the same type and \textit{type parameters} as ARRAY. The arguments and result shall not be \textit{polymorphic}. OPERATION should implement a mathematically associative operation. It need not be commutative.

DIM shall be an integer scalar with a value in the range \(1 \leq \text{DIM} \leq n\), where \(n\) is the rank of ARRAY.

MASK (optional) shall be of type logical and shall be \textit{conformable} with ARRAY.

IDENTITY (optional) shall be scalar with the same type and \textit{type parameters} as ARRAY.

ORDERED (optional) shall be a logical scalar.

4 Result Characteristics. The result is of the same type and \textit{type parameters} as ARRAY. It is scalar if DIM does not appear; otherwise, the result has rank \(n - 1\) and shape \([d_1, d_2, \ldots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \ldots, d_n]\) where \([d_1, d_2, \ldots, d_n]\) is the shape of ARRAY.

5 Result Value.

Case (i): The result of REDUCE (ARRAY, OPERATION [], IDENTITY = IDENTITY, ORDERED = ORDERED]) over the sequence of values in ARRAY is the result of an iterative process. The initial order of the sequence is array element order. While the sequence has more than one element, each iteration involves the execution of \(r = \text{OPERATION}(x, y)\) for adjacent \(x\) and \(y\) in the sequence, with \(x\) immediately preceding \(y\), and the subsequent replacement of \(x\) and \(y\) with \(r\); if ORDERED is present with the value true, \(x\) and \(y\) shall be the first two elements of the sequence. The process continues until the sequence has only one element which is the value of the reduction. If the initial sequence is empty, the result has the value IDENTITY if IDENTITY is present, and otherwise, error termination is initiated.

Case (ii): The result of REDUCE (ARRAY, OPERATION, MASK = MASK [], IDENTITY = IDENTITY, ORDERED = ORDERED]) is as for Case (i) except that the initial sequence is only those elements of ARRAY for which the corresponding elements of MASK are true.

Case (iii): If ARRAY has rank one, REDUCE (ARRAY, OPERATION, DIM = DIM [], MASK = MASK, IDENTITY = IDENTITY, ORDERED = ORDERED]) has a value equal to that of REDUCE (ARRAY, OPERATION [], MASK = MASK, IDENTITY = IDENTITY, ORDERED = ORDERED]).
Otherwise, the value of element \((s_1, s_2, \ldots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \ldots, s_n)\) of REDUCE \(\text{ARRAY}, \text{DIM} = \text{DIM}[, \text{MASK} = \text{MASK}, \text{IDENTITY} = \text{IDENTITY}]\) is equal to
\[
\text{REDUCE} \left( \text{ARRAY} \left( s_1, s_2, \ldots, s_{\text{DIM}-1}; s_{\text{DIM}+1}, \ldots, s_n \right), \text{OPERATION} = \text{OPERATION}, \text{DIM}=1 [, \text{MASK} = \text{MASK} (s_1, s_2, \ldots, s_{\text{DIM}-1}; s_{\text{DIM}+1}, \ldots, s_n), \text{IDENTITY} = \text{IDENTITY}, \text{ORDERED} = \text{ORDERED} ] \right).
\]

### 6 Examples.
The following examples all use the function MY\_MULT, which returns the product of its two integer arguments.

- **Case (i):** The value of REDUCE ([1, 2, 3], MY\_MULT) is 6.
- **Case (ii):** REDUCE (C, MY\_MULT, MASK = C > 0, IDENTITY=1) forms the product of the positive elements of C.
- **Case (iii):** If B is the array \[
\begin{bmatrix}
1 & 3 & 5 \\
2 & 4 & 6
\end{bmatrix}
\]
  REDUCE (B, MY\_MULT, DIM = 1) is [2, 12, 30] and REDUCE (B, MY\_MULT, DIM = 2) is [15, 48].

**NOTE 16.24**
If OPERATION is not computationally associative, REDUCE without ORDERED=.TRUE. with the same argument values might not always produce the same result, as the processor can apply the associative law to the evaluation.

### 16.9.162 REPEAT (STRING, NCOPIES)

#### Description.
Repetitive string concatenation.

#### Class.
Transformational function.

#### Arguments.
- **STRING** shall be a character scalar.
- **NCOPIES** shall be an integer scalar. Its value shall not be negative.

#### Result Characteristics.
Character scalar of length NCOPIES times that of STRING, with the same kind type parameter as STRING.

#### Result Value.
The value of the result is the concatenation of NCOPIES copies of STRING.

#### Examples.
REPEAT (’H’, 2) has the value HH. REPEAT (’XYZ’, 0) has the value of a zero-length string.

### 16.9.163 RESHAPE (SOURCE, SHAPE [, PAD, ORDER])

#### Description.
Arbitrary shape array construction.

#### Class.
Transformational function.

#### Arguments.
- **SOURCE** shall be an array of any type. If PAD is absent or of size zero, the size of SOURCE shall be greater than or equal to PRODUCT (SHAPE). The size of the result is the product of the values of the elements of SHAPE.
- **SHAPE** shall be a rank-one integer array. SIZE \((x)\), where \(x\) is the actual argument corresponding to SHAPE, shall be a constant expression whose value is positive and less than 16. It shall not have an element whose value is negative.
- **PAD** (optional) shall be an array of the same type and type parameters as SOURCE.
ORDER (optional) shall be of type integer, shall have the same shape as SHAPE, and its value shall be a permutation of \((1, 2, \ldots, n)\), where \(n\) is the size of SHAPE. If absent, it is as if it were present with value \((1, 2, \ldots, n)\).

**Result Characteristics.** The result is an array of shape SHAPE (that is, SHAPE (RESHAPE (SOURCE, SHAPE, PAD, ORDER)) is equal to SHAPE) with the same type and type parameters as SOURCE.

**Result Value.** The elements of the result, taken in permuted subscript order ORDER (1), ..., ORDER (n), are those of SOURCE in normal array element order followed if necessary by those of PAD in array element order, followed if necessary by additional copies of PAD in array element order.

**Examples.**

\[
\begin{bmatrix}
1 & 3 & 5 \\
2 & 4 & 6
\end{bmatrix}
\]

RESHAPE ([1, 2, 3, 4, 5, 6], [2, 3], [0, 0], [2, 1]) has the value

\[
\begin{bmatrix}
1 & 2 & 3 & 4 \\
5 & 6 & 0 & 0
\end{bmatrix}
\]

### 16.9.164 RRSPACING (X)

**Description.** Reciprocal of relative spacing of model numbers.

**Class.** Elemental function.

**Argument.** X shall be of type real.

**Result Characteristics.** Same as X.

**Result Value.** The result has the value

\[
|y_\times b^{-e} \times b^p| \times b^p = \text{ABS (FRACTION}\ (Y)) \times \text{RADIX}\ (X) / \text{EPSILON}\ (X),
\]

where \(b, e,\) and \(p\) are as defined in 16.4 for \(Y,\) the value nearest to \(X\) in the model for real values whose kind type parameter is that of \(X;\) if there are two such values, the value of greater absolute value is taken. If \(X\) is an IEEE infinity, the result is an IEEE NaN. If \(X\) is an IEEE NaN, the result is that NaN.

**Example.** RRSPACING \((-3.0)\) has the value \(0.75 \times 2^{24}\) for reals whose model is as in Note 16.4.

### 16.9.165 SAME_TYPE_AS (A, B)

**Description.** Dynamic type equality test.

**Class.** Inquiry function.

**Arguments.**

- **A** shall be an object of extensible declared type or unlimited polymorphic. If it is a polymorphic pointer, it shall not have an undefined association status.
- **B** shall be an object of extensible declared type or unlimited polymorphic. If it is a polymorphic pointer, it shall not have an undefined association status.

**Result Characteristics.** Default logical scalar.

**Result Value.** If the dynamic type of \(A\) or \(B\) is extensible, the result is true if and only if the dynamic type of \(A\) is the same as the dynamic type of \(B.\) If neither \(A\) nor \(B\) has extensible dynamic type, the result is processor dependent.

**Note 16.25**

The dynamic type of a disassociated pointer or unallocated allocatable variable is its declared type. An unlimited polymorphic entity has no declared type.

**Example.** Given the declarations and assignments

\[
\text{TYPE T1}
\]
REAL C
END TYPE

TYPE, EXTENDS(T1) :: T2
END TYPE

CLASS(T1), POINTER :: P, Q, R
ALLOCATE(P, Q)
ALLOCATE(T2 :: R)

the value of SAME_TYPE_AS (P, Q) will be true, and the value of SAME_TYPE_AS (P, R) will be false.

16.9.166 SCALE (X, I)

Description. Real number scaled by radix power.

Class. Elemental function.

Arguments.

X shall be of type real.
I shall be of type integer.

Result Characteristics. Same as X.

Result Value. The result has the value \( X \times b^I \), where \( b \) is defined in 16.4 for model numbers representing values of \( X \), provided this result is representable; if not, the result is processor dependent.

Example. SCALE (3.0, 2) has the value 12.0 for reals whose model is as in Note 16.4.

16.9.167 SCAN (STRING, SET [, BACK, KIND])

Description. Character set membership search.

Class. Elemental function.

Arguments.

STRING shall be of type character.
SET shall be of type character with the same kind type parameter as STRING.
BACK (optional) shall be of type logical.
KIND (optional) shall be a scalar integer constant expression.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type.

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.

16.9.168 SELECTED_CHAR_KIND (NAME)

Description. Character kind selection.

Class. Transformational function.

Argument. NAME shall be default character scalar.

Result Characteristics. Default integer scalar.

Result Value. If NAME has the value DEFAULT, then the result has a value equal to that of the kind type parameter of default character. If NAME has the value ASCII, then the result has a value equal to that of the kind type parameter of ASCII character if the processor supports such a kind; otherwise the result has the value \(-1\). If NAME has the value ISO_10646, then the result has a value equal to that of the kind type parameter of the ISO 10646 character kind (corresponding to UCS-4 as specified in ISO/IEC 10646) if the processor supports such a kind; otherwise the result has the value \(-1\). If NAME is a processor-defined name of some other character kind supported by the processor, then the result has a value equal to that kind type parameter value. If NAME is not the name of a supported character type, then the result has the value \(-1\). The NAME is interpreted without respect to case or trailing blanks.

Examples. SELECTED_CHAR_KIND (‘ASCII’) has the value 1 on a processor that uses 1 as the kind type parameter for the ASCII character set. The following subroutine produces a Japanese date stamp.

```
SUBROUTINE create_date_string(string)
  INTRINSIC date_and_time,selected_char_kind
  INTEGER,PARAMETER :: ucs4 = selected_char_kind("ISO_10646")
  CHARACTER(1,UCS4),PARAMETER :: nen=CHAR(INT(Z’5e74’),UCS4), & !year
    gatsu=CHAR(INT(Z’6708’),UCS4), & !month
    nichi=CHAR(INT(Z’65e5’),UCS4) !day
  CHARACTER(len= *, kind= ucs4) string
  INTEGER values(8)
  CALL date_and_time(values=values)
  WRITE(string,1) values(1),nen,values(2),gatsu,values(3),nichi
  1 FORMAT(I0,A,I0,A,I0,A)
END SUBROUTINE
```

16.9.169 SELECTED_INT_KIND (R)

Description. Integer kind selection.

Class. Transformational function.

Argument. R shall be an integer scalar.

Result Characteristics. Default integer scalar.

Result Value. The result has a value equal to the value of the kind type parameter of an integer type that represents all values \(n\) in the range \(-10^R < n < 10^R\), or if no such kind type parameter is available on the processor, the result is \(-1\). If more than one kind type parameter meets the criterion, the value returned is the one with the smallest decimal exponent range, unless there are several such values, in which case the smallest of these kind values is returned.

Example. Assume a processor supports two integer kinds, 32 with representation method \(r = 2\) and \(q = 31\), and 64 with representation method \(r = 2\) and \(q = 63\). On this processor SELECTED_INT_KIND (9) has the...
value 32 and SELECTED_INT_KIND (10) has the value 64.

16.9.170 SELECTED_REAL_KIND ([P, R, RADIX])

1 Description. Real kind selection.
2 Class. Transformational function.
3 Arguments. At least one argument shall be present.
4 P (optional) shall be an integer scalar.
5 R (optional) shall be an integer scalar.
6 RADIX (optional) shall be an integer scalar.

4 Result Characteristics. Default integer scalar.
5 Result Value. If P or R is absent, the result value is the same as if it were present with the value zero. If RADIX is absent, there is no requirement on the radix of the selected kind.
6 The result has a value equal to a value of the kind type parameter of a real type with decimal precision, as returned by the function PRECISION, of at least P digits, a decimal exponent range, as returned by the function RANGE, of at least R, and a radix, as returned by the function RADIX, of RADIX, if such a kind type parameter is available on the processor.
7 Otherwise, the result is $-1$ if the processor supports a real type with radix RADIX and exponent range of at least R but not with precision of at least P, $-2$ if the processor supports a real type with radix RADIX and precision of at least P but not with exponent range of at least R, $-3$ if the processor supports a real type with radix RADIX but with neither precision of at least P nor exponent range of at least R, $-4$ if the processor supports a real type with radix RADIX and either precision of at least P or exponent range of at least R but not both together, and $-5$ if the processor supports no real type with radix RADIX.
8 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.

9 Example. SELECTED_REAL_KIND (6, 70) has the value KIND (0.0) on a machine that supports a default real approximation method with $b = 16$, $p = 6$, $e_{\text{min}} = -64$, and $e_{\text{max}} = 63$ and does not have a less precise approximation method.

16.9.171 SET_EXPONENT (X, I)

1 Description. Real value with specified exponent.
2 Class. Elemental function.
3 Arguments.
4 X shall be of type real.
5 I shall be of type integer.

4 Result Characteristics. Same as X.
5 Result Value. If X has the value zero, the result has the same value as X. If X is an IEEE infinity, the result is an IEEE NaN. If X is an IEEE NaN, the result is the same NaN. Otherwise, the result has the value $X \times b^{i-e}$, where $b$ and $e$ are as defined in 16.4 for the representation for the value of X in the extended real model for the kind of X.

6 Example. SET_EXPONENT (3.0, 1) has the value 1.5 for reals whose model is as in Note 16.4.
16.9.172 SHAPE ([SOURCE], [KIND])

Description. Shape of an array or a scalar.

Class. Inquiry function.

Arguments.
SOURCE shall be a scalar or array of any type. It shall not be an unallocated allocatable variable or a pointer that is not associated. It shall not be an assumed-size array.

KIND (optional) shall be a scalar integer constant expression.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type. The result is an array of rank one whose size is equal to the rank of SOURCE.

Result Value. The result has a value whose \( i \)th element is equal to the extent of dimension \( i \) of SOURCE, except that if SOURCE is assumed-rank, and associated with an assumed-size array, the last element is equal to \(-1\).

Examples. The value of SHAPE (A (2:5, \(-1:1\) ) ) is \([4, 3]\). The value of SHAPE (3) is the rank-one array of size zero.

16.9.173 SHIFTA (I, SHIFT)

Description. Right shift with fill.

Class. Elemental function.

Arguments.
I shall be of type integer.

SHIFT shall be of type integer. It shall be nonnegative and less than or equal to BIT_SIZE (I).

Result Characteristics. Same as I.

Result Value. The result has the value obtained by shifting the bits of I to the right SHIFT bits and replicating the leftmost bit of I in the left SHIFT bits.

If SHIFT is zero the result is I. Bits shifted out from the right are lost. The model for the interpretation of an integer value as a sequence of bits is in 16.3.

Example. SHIFTA (IBSET (0, BIT_SIZE (0) \(-1\) ), 2) is equal to SHIFTL (7, BIT_SIZE (0) \(-3\) ).

16.9.174 SHIFTL (I, SHIFT)

Description. Left shift.

Class. Elemental function.

Arguments.
I shall be of type integer.

SHIFT shall be of type integer. It shall be nonnegative and less than or equal to BIT_SIZE (I).

Result Characteristics. Same as I.

Result Value. The value of the result is ISHFT (I, SHIFT).

Examples. SHIFTL (3, 1) has the value 6.
16.9.175 SHIFTR (I, SHIFT)

Description. Right shift.

Class. Elemental function.

Arguments.

I shall be of type integer.
SHIFT shall be of type integer. It shall be nonnegative and less than or equal to BIT_SIZE (I).

Result Characteristics. Same as I.

Result Value. The value of the result is ISHFT (I, -SHIFT).

Examples. SHIFTR (3, 1) has the value 1.

16.9.176 SIGN (A, B)

Description. Magnitude of A with the sign of B.

Class. Elemental function.

Arguments.

A shall be of type integer or real.
B shall be of the same type as A.

Result Characteristics. Same as A.

Result Value.

Case (i): If B > 0, the value of the result is |A|.
Case (ii): If B < 0, the value of the result is -|A|.
Case (iii): If B is of type integer and B=0, the value of the result is |A|.
Case (iv): If B is of type real and is zero, then:

- if the processor does not distinguish between positive and negative real zero, or if B is positive real zero, the value of the result is |A|;
- if the processor distinguishes between positive and negative real zero, and B is negative real zero, the value of the result is -|A|.

Example. SIGN (-3.0, 2.0) has the value 3.0.

16.9.177 SIN (X)

Description. Sine function.

Class. Elemental function.

Argument. X shall be of type real or complex.

Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to sin(X). If X is of type real, it is regarded as a value in radians. If X is of type complex, its real part is regarded as a value in radians.

Example. SIN (1.0) has the value 0.84147098 (approximately).
16.9.178  **SINH (X)**

1 **Description.** Hyperbolic sine function.

2 **Class.** Elemental function.

3 **Argument.** X shall be of type real or complex.

4 **Result Characteristics.** Same as X.

5 **Result Value.** The result has a value equal to a processor-dependent approximation to \( \sinh(X) \). If X is of type complex its imaginary part is regarded as a value in radians.

6 **Example.** SINH (1.0) has the value 1.1752012 (approximately).

16.9.179  **SIZE (ARRAY [, DIM, KIND])**

1 **Description.** Size of an array or one extent.

2 **Class.** Inquiry function.

3 **Arguments.**

4 **ARRAY** shall be assumed-rank or an array. It shall not be an unallocated allocatable variable or a pointer that is not associated. If ARRAY is an assumed-size array, DIM shall be present with a value less than the rank of ARRAY.

5 **DIM** (optional) shall be an integer scalar with a value in the range \( 1 \leq \text{DIM} \leq n \), where \( n \) is the rank of ARRAY.

6 **KIND** (optional) shall be a scalar integer constant expression.

7 **Result Characteristics.** Integer scalar. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type.

8 **Result Value.** If DIM is present, the result has a value equal to the extent of dimension DIM of ARRAY, except that if ARRAY is assumed-rank and associated with an assumed-size array and DIM is present with a value equal to the rank of ARRAY, the value is \(-1\).

9 If DIM is absent and ARRAY is assumed-rank, the result has a value equal to \( \text{PRODUCT}(	ext{SHAPE}(\text{ARRAY}, \text{KIND})) \). Otherwise, the result has a value equal to the total number of elements of ARRAY.

10 **Examples.** The value of SIZE (A (2:5, -1:1), DIM=2) is 3. The value of SIZE (A (2:5, -1:1)) is 12.

**NOTE 16.26**

If ARRAY is assumed-rank and has rank zero, DIM cannot be present since it cannot satisfy the requirement \( 1 \leq \text{DIM} \leq 0 \).

16.9.180  **SPACING (X)**

1 **Description.** Spacing of model numbers.

2 **Class.** Elemental function.

3 **Argument.** X shall be of type real.

4 **Result Characteristics.** Same as X.

5 **Result Value.** If X does not have the value zero and is not an IEEE infinity or NaN, the result has the value \( b^{\text{e}^p} \), where \( b, e, \) and \( p \) are as defined in 16.4 for the value nearest to X in the model for real values whose kind type parameter is that of X, provided this result is representable; otherwise, the result is the same as that of TINY (X). If there are two extended model values equally near to X, the value of greater absolute value is taken.
If X has the value zero, the result is the same as that of TINY (X). If X is an IEEE infinity, the result is an IEEE NaN. If X is an IEEE NaN, the result is that NaN.

Example. SPACING (3.0) has the value $2^{-22}$ for reals whose model is as in Note 16.4.

16.9.181 SPREAD (SOURCE, DIM, NCOPIES)

1 Description. Value replicated in a new dimension.
2 Class. Transformational function.
3 Arguments.
   SOURCE shall be a scalar or array of any type. The rank of SOURCE shall be less than 15.
   DIM shall be an integer scalar with value in the range $1 \leq DIM \leq n+1$, where $n$ is the rank of SOURCE.
   NCOPIES shall be an integer scalar.
4 Result Characteristics. The result is an array of the same type and type parameters as SOURCE and of rank $n+1$, where $n$ is the rank of SOURCE.
   Case (i): If SOURCE is scalar, the shape of the result is $(\text{MAX (NCOPIES, 0)})$.
   Case (ii): If SOURCE is an array with shape $[d_1, d_2, \ldots, d_n]$, the shape of the result is $[d_1, d_2, \ldots, d_{\text{DIM}-1}, \text{MAX (NCOPIES, 0)}, d_{\text{DIM}}, \ldots, d_n]$.
5 Result Value.
   Case (i): If SOURCE is scalar, each element of the result has a value equal to SOURCE.
   Case (ii): If SOURCE is an array, the element of the result with subscripts $(r_1, r_2, \ldots, r_{n+1})$ has the value SOURCE $(r_1, r_2, \ldots, r_{\text{DIM}-1}, r_{\text{DIM}+1}, \ldots, r_{n+1})$.
6 Examples. If A is the array $[2, 3, 4]$, SPREAD (A, DIM=1, NCOPIES=NC) is the array
   \[
   \begin{bmatrix}
   2 & 3 & 4 \\
   2 & 3 & 4 \\
   \end{bmatrix}
   \]
   if NC has the value 3 and is a zero-sized array if NC has the value 0.

16.9.182 SQRT (X)

1 Description. Square root.
2 Class. Elemental function.
3 Argument. X shall be of type real or complex. If X is real, its value shall be greater than or equal to zero.
4 Result Characteristics. Same as X.
5 Result Value. The result has a value equal to a processor-dependent approximation to the square root of X. A result of type complex is the principal value with the real part greater than or equal to zero. When the real part of the result is zero, the imaginary part has the same sign as the imaginary part of X.
6 Example. SQRT (4.0) has the value 2.0 (approximately).

16.9.183 STOPPED_IMAGES([TEAM, KIND])

1 Description. Indices of stopped images.
2 Class. Transformational function.
3 Arguments.
   TEAM (optional) shall be a scalar of type TEAM_TYPE from the intrinsic module ISO_FORTRAN_ENV, whose value identifies the current or an ancestor team. If TEAM is absent the team specified is the current team.
KIND (optional) shall be a scalar integer constant expression.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default integer type. The result is an array of rank one whose size is equal to the number of images in the specified team that have initiated normal termination.

Result Value. The elements of the result are the values of the indices of the images that are known to have initiated normal termination in the specified team, in numerically increasing order. If the executing image has previously executed an image control statement whose STAT= specifier assigned the value STAT_STOPPED_IMAGE from the intrinsic module ISO_FORTRAN_ENV or invoked a collective subroutine whose STAT argument was assigned STAT_STOPPED_IMAGE, at least one of the images participating in that image control statement or collective invocation shall be known to have initiated normal termination.

Examples. If image 3 is the only image in the current team that is known to have initiated normal termination, STOPPED_IMAGES() will have the value [3]. If there are no images in the current team that have initiated normal termination, the value of STOPPED_IMAGES() will be a zero-sized array.

16.9.184 STORAGE_SIZE (A [, KIND])

Description. Storage size in bits.

Class. Inquiry function.

Arguments.
A shall be a data object of any type. If it is polymorphic it shall not be an undefined pointer. If it is unlimited polymorphic or has any deferred type parameters, it shall not be an unallocated allocatable variable or a disassociated or undefined pointer.

KIND (optional) shall be a scalar integer constant expression.

Result Characteristics. Integer scalar. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default integer type.

Result Value. The result value is the size expressed in bits for an element of an array that has the dynamic type and type parameters of A. If the type and type parameters are such that storage association (19.5.3) applies, the result is consistent with the named constants defined in the intrinsic module ISO_FORTRAN_ENV.

NOTE 16.27
An array element might take more bits to store than an isolated scalar, since any hardware-imposed alignment requirements for array elements might not apply to a simple scalar variable.

NOTE 16.28
This is intended to be the size in memory that an object takes when it is stored; this might differ from the size it takes during expression handling (which might be the native register size) or when stored in a file. If an object is never stored in memory but only in a register, this function nonetheless returns the size it would take if it were stored in memory.

Example. STORAGE_SIZE (1.0) has the same value as the named constant NUMERIC_STORAGE_SIZE in the intrinsic module ISO_FORTRAN_ENV.

16.9.185 SUM (ARRAY, DIM [, MASK]) or SUM (ARRAY [, MASK])

Description. Array reduced by addition.

Class. Transformational function.

Arguments.
ARRAY shall be an array of numeric type.
DIM shall be an integer scalar with a value in the range $1 \leq \text{DIM} \leq n$, where $n$ is the rank of ARRAY.
MASK (optional) shall be of type logical and shall be conformable with ARRAY.

4 Result Characteristics. The result is of the same type and kind type parameter as ARRAY. It is scalar if DIM does not appear; otherwise, the result has rank $n - 1$ and shape $[d_1, d_2, \ldots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \ldots, d_n]$ where $[d_1, d_2, \ldots, d_n]$ is the shape of ARRAY.

5 Result Value.

Case (i): The result of SUM (ARRAY) has a value equal to a processor-dependent approximation to the sum of all the elements of ARRAY or has the value zero if ARRAY has size zero.
Case (ii): The result of SUM (ARRAY, MASK = MASK) has a value equal to a processor-dependent approximation to the sum of the elements of ARRAY corresponding to the true elements of MASK or has the value zero if there are no true elements.
Case (iii): If ARRAY has rank one, SUM (ARRAY, DIM = DIM [, MASK = MASK]) has a value equal to that of SUM (ARRAY [,MASK = MASK]). Otherwise, the value of element $(s_1, s_2, \ldots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \ldots, s_n)$ of SUM (ARRAY, DIM = DIM [, MASK = MASK]) is equal to

$$\text{SUM (ARRAY} (s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n) [, \text{MASK} = \text{MASK} (s_1, s_2, \ldots, s_{\text{DIM}-1}, \ldots, s_{\text{DIM}+1}, \ldots, s_n) ] \).$$

6 Examples.

Case (i): The value of SUM ([1, 2, 3]) is 6.
Case (ii): SUM (C, MASK = C > 0.0) forms the sum of the positive elements of C.
Case (iii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$, SUM (B, DIM = 1) is [3, 7, 11] and SUM (B, DIM = 2) is [9, 12].

16.9.186 SYSTEM_CLOCK ([COUNT, COUNT_RATE, COUNT_MAX])

1 Description. Query system clock.
2 Class. Subroutine.
3 Arguments.

COUNT (optional) shall be an integer scalar. It is an INTENT (OUT) argument. It is assigned a processor-dependent value based on the value of a processor clock, or $-\text{HUGE (COUNT)}$ if there is no clock for the invoking image. The processor-dependent value is incremented by one for each clock count until the value COUNT_MAX is reached and is reset to zero at the next count. It lies in the range 0 to COUNT_MAX if there is a clock.
COUNT_RATE (optional) shall be an integer or real scalar. It is an INTENT (OUT) argument. It is assigned a processor-dependent approximation to the number of processor clock counts per second, or zero if there is no clock for the invoking image.
COUNT_MAX (optional) shall be an integer scalar. It is an INTENT (OUT) argument. It is assigned the maximum value that COUNT can have, or zero if there is no clock for the invoking image.

4 Whether an image has no clock, has a single clock of its own, or shares a clock with another image, is processor dependent.

5 Example. If the processor clock is a 24-hour clock that registers time at approximately 18.20648193 ticks per second, at 11:30 A.M. the reference

$$\text{CALL SYSTEM_CLOCK (COUNT = C, COUNT_RATE = R, COUNT_MAX = M)}$$
defines $C = (11 \times 3600 + 30 \times 60) \times 18.20648193 = 753748$, $R = 18.20648193$, and $M = 24 \times 3600 \times 18.20648193 - 1 = 1573039$. 

ISO/IEC JTC 1/SC 22/WG5/N2118 433
16.9.187 TAN (X)

**Description.** Tangent function.

**Class.** Elemental function.

**Argument.** X shall be of type real or complex.

**Result Characteristics.** Same as X.

**Result Value.** The result has a value equal to a processor-dependent approximation to \( \tan(X) \). If X is of type real, it is regarded as a value in radians. If X is of type complex, its real part is regarded as a value in radians.

**Example.** \( \text{TAN} (1.0) \) has the value 1.5574077 (approximately).

16.9.188 TANH (X)

**Description.** Hyperbolic tangent function.

**Class.** Elemental function.

**Argument.** X shall be of type real or complex.

**Result Characteristics.** Same as X.

**Result Value.** The result has a value equal to a processor-dependent approximation to \( \tanh(X) \). If X is of type complex its imaginary part is regarded as a value in radians.

**Example.** \( \text{TANH} (1.0) \) has the value 0.76159416 (approximately).

16.9.189 TEAM_NUMBER([TEAM])

**Description.** Team number.

**Class.** Transformational function.

**Argument.** TEAM (optional) shall be a scalar of type TEAM_TYPE from the intrinsic module ISO_FORTRAN_ENV, whose value identifies the current or an ancestor team. If TEAM is absent, the team specified is the current team.

**Result Characteristics.** Default integer scalar.

**Result Value.** If the specified team is the initial team, the result value is \(-1\); otherwise, the result value is the team number identifying the team of the invoking image within the specified team.

**Example.** The team number can be used to control which statements get executed, for example:

```fortran
TYPE(TEAM_TYPE) :: ODD_EVEN

... FORM TEAM (2-MOD(ME,2), ODD_EVEN)

... CHANGE TEAM (ODD_EVEN)

SELECT CASE (TEAM_NUMBER())
CASE (1)
! Case for images with odd image indices in the parent team.
CASE (2)
! Case for images with even image indices in the parent team.
END SELECT
```
16.9.190 **THIS_IMAGE ([TEAM]) or THIS_IMAGE (COARRAY [, TEAM]) or THIS_IMAGE (COARRAY, DIM [, TEAM])**

1 Description. Cosubscript(s) for this image.

2 Class. Transformational function.

3 Arguments. 
   COARRAY shall be a coarray of any type. If it is allocatable it shall be allocated. If it is of type TEAM_TYPE from the intrinsic module ISO_FORTRAN_ENV, the TEAM argument shall appear.
   
   DIM shall be an integer scalar. Its value shall be in the range $1 \leq \text{DIM} \leq n$, where $n$ is the corank of COARRAY.
   
   TEAM (optional) shall be a scalar of type TEAM_TYPE from the intrinsic module ISO_FORTRAN_ENV, whose value identifies the current or an ancestor team. If COARRAY appears, it shall be established in that team.

4 Result Characteristics. Default integer. It is scalar if COARRAY does not appear or DIM appears; otherwise, the result has rank one and its size is equal to the corank of COARRAY.

5 Result Value. 
   
   Case (i): The result of THIS_IMAGE ([TEAM]) is a scalar with a value equal to the index of the invoking image in the team specified by TEAM, if present, or in the current team if absent.
   
   Case (ii): The result of THIS_IMAGE (COARRAY [, TEAM = TEAM]) is the sequence of cosubscript values for COARRAY that would specify the invoking image in the team specified by TEAM, if present, or in the current team if absent.
   
   Case (iii): The result of THIS_IMAGE (COARRAY, DIM [, TEAM = TEAM]) is the value of cosubscript DIM in the sequence of cosubscript values for COARRAY that would specify the invoking image in the team specified by TEAM, if present, or in the current team if absent.

6 Examples. If A is declared by the statement
   
   ```fortran
   REAL A (10, 20) [10, 0:9, 0:*]
   ```
   then on image 5, THIS_IMAGE ( ) has the value 5 and THIS_IMAGE (A) has the value [5, 0, 0]. For the same coarray on image 213, THIS_IMAGE (A) has the value [3, 1, 2].
   
   The following code uses image 1 to read data. The other images then copy the data.
   
   ```fortran
   IF (THIS_IMAGE()==1) READ (*,*) P
   SYNC ALL
   P = P[1]
   ```

16.9.191 **TINY (X)**

1 Description. Smallest positive model number.

2 Class. Inquiry function.

3 Argument. X shall be a real scalar or array.

4 Result Characteristics. Scalar with the same type and kind type parameter as X.

5 Result Value. The result has the value $b^{-\epsilon_{\text{min}}}$ where $b$ and $\epsilon_{\text{min}}$ are as defined in 16.4 for the model representing numbers of the same type and kind type parameter as X.

6 Example. TINY (X) has the value $2^{-127}$ for real X whose model is as in Note 16.4.
16.9.192 TRAILZ (I)

Description. Number of trailing zero bits.

Class. Elemental function.

Argument. I shall be of type integer.

Result Characteristics. Default integer.

Result Value. If all of the bits of I are zero, the result value is BIT_SIZE (I). Otherwise, the result value is the position of the rightmost 1 bit in I. The model for the interpretation of an integer value as a sequence of bits is in 16.3.

Examples. TRAILZ (8) has the value 3.

16.9.193 TRANSFER (SOURCE, MOLD [, SIZE])

Description. Transfer physical representation.

Class. Transformational function.

Arguments. SOURCE shall be a scalar or array of any type.

MOLD shall be a scalar or array of any type. If it is a variable, it need not be defined. If the storage size of SOURCE is greater than zero and MOLD is an array, a scalar with the type and type parameters of MOLD shall not have a storage size equal to zero.

SIZE (optional) shall be an integer scalar. The corresponding actual argument shall not be an optional dummy argument.

Result Characteristics. The result is of the same type and type parameters as MOLD.

Case (i): If MOLD is a scalar and SIZE is absent, the result is a scalar.

Case (ii): If MOLD is an array and SIZE is absent, the result is an array and of rank one. Its size is as small as possible such that its physical representation is not shorter than that of SOURCE.

Case (iii): If SIZE is present, the result is an array of rank one and size SIZE.

Result Value. If the physical representation of the result has the same length as that of SOURCE, the physical representation of the result is that of SOURCE. If the physical representation of the result is longer than that of SOURCE, the physical representation of the leading part is that of SOURCE and the remainder is processor dependent. If the physical representation of the result is shorter than that of SOURCE, the physical representation of the remainder is processor dependent. If D and E are scalar variables such that the physical representation of D is as long as or longer than that of E, the value of TRANSFER (TRANSFER (E, D), E) shall be the value of E. If D is an array and E is an array of rank one, the value of TRANSFER (TRANSFER (E, D), E, SIZE (E)) shall be the value of E.

Examples.

Case (i): TRANSFER (1082130432, 0.0) has the value 4.0 on a processor that represents the values 4.0 and 1082130432 as the string of binary digits 0100 0000 1000 0000 0000 0000 0000 0000.

Case (ii): 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 processor dependent.

Case (iii): TRANSFER ([1.1, 2.2, 3.3], [(0.0, 0.0)], 1) is a complex rank-one array of length one whose only element has the value (1.1, 2.2).
16.9.194 **TRANPOSE (MATRIX)**

1. **Description.** Transpose of an array of rank two.
2. **Class.** Transformational function.
3. **Argument.** MATRIX shall be a rank-two array of any type.
4. **Result Characteristics.** The result is an array of the same type and type parameters as MATRIX and with rank two and shape \([n, m]\) where \([m, n]\) is the shape of MATRIX.
5. **Result Value.** Element \((i, j)\) of the result has the value \(MATRIX(j + LBOUND(MATRIX, 1) - 1, i + LBOUND(MATRIX, 2) - 1)\).

6. **Example.** If \(A\) is the array

\[
\begin{pmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{pmatrix}
\]

then \(\text{TRANSPOSE}(A)\) has the value

\[
\begin{pmatrix}
1 & 4 & 7 \\
2 & 5 & 8 \\
3 & 6 & 9
\end{pmatrix}
\]

16.9.195 **TRIM (STRING)**

1. **Description.** String without trailing blanks.
2. **Class.** Transformational function.
3. **Argument.** STRING shall be a character scalar.
4. **Result Characteristics.** 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. If STRING contains no nonblank characters, the result has zero length.
5. **Result Value.** The value of the result is the same as STRING except any trailing blanks are removed.
6. **Example.** \(\text{TRIM}(' A B ')\) has the value 'A B'.

16.9.196 **UBOUND (ARRAY [, DIM, KIND])**

1. **Description.** Upper bound(s).
2. **Class.** Inquiry function.
3. **Arguments.**
   - ARRAY shall be assumed-rank or an array. It shall not be an unallocated allocatable array or a pointer that is not associated. If ARRAY is an assumed-size array, DIM shall be present with a value less than the rank of ARRAY.
   - DIM (optional) shall be an integer scalar with a value in the range \(1 \leq \text{DIM} \leq n\), where \(n\) is the rank of ARRAY. The corresponding actual argument shall not be an optional dummy argument, a disassociated pointer, or an unallocated allocatable.
   - KIND (optional) shall be a scalar integer constant expression.
4. **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type. The result is scalar if DIM is present; otherwise, the result is an array of rank one and size \(n\), where \(n\) is the rank of ARRAY.
5. **Result Value.**

Case (i): If DIM is present, ARRAY is a whole array, and dimension DIM of ARRAY has nonzero extent, the result has a value equal to the upper bound for subscript DIM of ARRAY. Otherwise, if DIM is present and ARRAY is assumed-rank, the value of the result is as if ARRAY were a whole array, with the extent of the final dimension of ARRAY when ARRAY is associated with an assumed-size
array being considered to be $-1$. Otherwise, if DIM is present, the result has a value equal to the number of elements in dimension DIM of ARRAY.

Case (ii): If ARRAY has rank zero, UBOUND (ARRAY) has a value that is a zero-sized array. Otherwise, UBOUND (ARRAY) has a value whose $i^{th}$ element is equal to UBOUND (ARRAY, $i$), for $i = 1, 2, \ldots, n$, where $n$ is the rank of ARRAY. UBOUND (ARRAY, KIND=KIND) has a value whose $i^{th}$ element is equal to UBOUND (ARRAY, $i$, KIND=KIND), for $i = 1, 2, \ldots, n$, where $n$ is the rank of ARRAY.

**Examples.** If A is declared by the statement

```fortran
REAL A (2:3, 7:10)
```

then UBOUND (A) is [3, 10] and UBOUND (A, DIM = 2) is 10.

**NOTE 16.29**

If ARRAY is assumed-rank and has rank zero, DIM cannot be present since it cannot satisfy the requirement $1 \leq \text{DIM} \leq 0$.

### 16.9.197 UCOBOUND (COARRAY [, DIM, KIND])

**Description.** Upper cobound(s) of a coarray.

**Class.** Inquiry function.

**Arguments.**

- COARRAY shall be a coarray of any type. It may be a scalar or an array. If it is allocatable it shall be allocated.
- DIM (optional) shall be an integer scalar with a value in the range $1 \leq \text{DIM} \leq n$, where $n$ is the corank of COARRAY. The corresponding actual argument shall not be an optional dummy argument, a disassociated pointer, or an unallocated allocatable.
- KIND (optional) shall be a scalar integer constant expression.

**Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default integer type. The result is scalar if DIM is present; otherwise, the result is an array of rank one and size $n$, where $n$ is the corank of COARRAY.

**Result Value.** The final upper cobound is the final cosubscript in the cosubscript list for the coarray that selects the image whose index is equal to the number of images in the team current when COARRAY was established.

Case (i): If DIM is present, the result has a value equal to the upper cobound for codimension DIM of COARRAY.

Case (ii): If DIM is absent, the result has a value whose $i^{th}$ element is equal to the upper cobound for codimension $i$ of COARRAY, for $i = 1, 2, \ldots, n$, where $n$ is the corank of COARRAY.

**Examples.** If NUM IMAGES( ) has the value 30 and A is allocated by the statement

```fortran
ALLOCATE (A [2:3, 0:7, +1])
```

then UCOBOUND (A) is [3, 7, 2] and UCOBOUND (A, DIM=2) is 7. Note that the cosubscripts [3, 7, 2] do not correspond to an actual image.

### 16.9.198 UNPACK (VECTOR, MASK, FIELD)

**Description.** Vector unpacked into an array.

**Class.** Transformational function.

**Arguments.**

- VECTOR shall be a rank-one array of any type. Its size shall be at least $t$ where $t$ is the number of true elements in MASK.
MASK shall be a logical array.
FIELD shall be of the same type and type parameters as VECTOR and shall be conformable with MASK.

4 Result Characteristics. The result is an array of the same type and type parameters as VECTOR and the same shape as MASK.

5 Result Value. The element of the result that corresponds to the $i^{th}$ true element of MASK, in array element order, has the value VECTOR ($i$) for $i = 1, 2, \ldots, t$, where $t$ is the number of true values in MASK. Each other element has a value equal to FIELD if FIELD is scalar or to the corresponding element of FIELD if it is an array.

6 Examples. Particular values can be “scattered” to particular positions in an array by using UNPACK. If M is the array
$$
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1 \\
\end{bmatrix}
$$
V is the array [1, 2, 3], and Q is the logical mask
$$
\begin{bmatrix}
T & . \\
T & . \\
. & T \\
\end{bmatrix}
$$
where “T” represents true and “.” represents false, then the result of UNPACK (V, MASK = Q, FIELD = M) has the value
$$
\begin{bmatrix}
1 & 2 & 0 \\
1 & 1 & 0 \\
0 & 0 & 3 \\
\end{bmatrix}
$$
and the result of UNPACK (V, MASK = Q, FIELD = 0) has the value
$$
\begin{bmatrix}
0 & 2 & 0 \\
1 & 0 & 0 \\
0 & 0 & 3 \\
\end{bmatrix}
$$

16.9.199 VERIFY (STRING, SET [, BACK, KIND])

1 Description. Character set non-membership search.

2 Class. Elemental function.

3 Arguments.
STRING shall be of type character.
SET shall be of type character with the same kind type parameter as STRING.
BACK (optional) shall be of type logical.
KIND (optional) shall be a scalar integer constant expression.

4 Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type.

5 Result Value.
Case (i): If BACK is absent or has 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.

6 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.
16.10 Standard intrinsic modules

16.10.1 General

1 This document defines five standard intrinsic modules: a Fortran environment module, a set of three modules to support floating-point exceptions and IEEE arithmetic, and a module to support interoperability with the C programming language.

2 The intrinsic modules IEEE_EXCEPTIONS, IEEE_ARITHMETIC, and IEEE_FEATURES are described in Clause 17. The intrinsic module ISO_C_BINDING is described in Clause 18. The module procedures described in 16.10.2 are pure.

NOTE 16.30 The types and procedures defined in standard intrinsic modules are not themselves intrinsic.

3 A processor may extend the standard intrinsic modules to provide public entities in them in addition to those specified in this document.

16.10.2 The ISO_FORTRAN_ENV intrinsic module

16.10.2.1 General

1 The intrinsic module ISO_FORTRAN_ENV provides public entities relating to the Fortran environment.

2 The processor shall provide the named constants, derived types, and procedures described in subclause 16.10.2. In the detailed descriptions below, procedure names are generic and not specific.

16.10.2.2 ATOMIC_INT_KIND

1 The value of the default integer scalar constant ATOMIC_INT_KIND is the kind type parameter value of type integer variables for which the processor supports atomic operations specified by atomic subroutines.

16.10.2.3 ATOMIC_LOGICAL_KIND

1 The value of the default integer scalar constant ATOMIC_LOGICAL_KIND is the kind type parameter value of type logical variables for which the processor supports atomic operations specified by atomic subroutines.

16.10.2.4 CHARACTER_KINDS

1 The values of the elements of the default integer array constant CHARACTER_KINDS are the kind values supported by the processor for variables of type character. The order of the values is processor dependent. The rank of the array is one, its lower bound is one, and its size is the number of character kinds supported.

16.10.2.5 CHARACTER_STORAGE_SIZE

1 The value of the default integer scalar constant CHARACTER_STORAGE_SIZE is the size expressed in bits of the character storage unit (19.5.3.2).

16.10.2.6 COMPILER_OPTIONS ( )

1 Description. Processor-dependent string describing the options that controlled the program translation phase.

2 Class. Transformational function.

3 Argument. None.
4 **Result Characteristics.** Default character scalar with processor-dependent length.
5 **Result Value.** A processor-dependent value which describes the options that controlled the translation phase of program execution. This value should include relevant information that could be useful for diagnosing problems at a later date.

6 **Example.** COMPILER\_OPTIONS ( ) might have the value '/OPTIMIZE /FLOAT=IEEE'.

16.10.2.7 **COMPILER\_VERSION ( )**

1 **Description.** Processor-dependent string identifying the program translation phase.
2 **Class.** Transformational function.
3 **Argument.** None.

4 **Result Characteristics.** Default character scalar with processor-dependent length.
5 **Result Value.** A processor-dependent value that identifies the name and version of the program translation phase of the processor. This value should include relevant information that could be useful for diagnosing problems at a later date.

6 **Example.** COMPILER\_VERSION ( ) might have the value 'Fast KL-10 Compiler Version 7'.

**NOTE 16.31**

Relevant information that could be useful for diagnosing problems at a later date might include compiler release and patch level, default compiler arguments, environment variable values, and run time library requirements. A processor might include this information in an object file automatically, without the user needing to save the result of this function in a variable.

16.10.2.8 **CURRENT\_TEAM**

1 The value of the default integer scalar constant CURRENT\_TEAM identifies the current team when it is used as the LEVEL argument to GET\_TEAM.

16.10.2.9 **ERROR\_UNIT**

1 The value of the default integer scalar constant ERROR\_UNIT identifies the processor-dependent preconnected external unit used for the purpose of error reporting (12.5). This unit may be the same as OUTPUT\_UNIT. The value shall not be −1.

16.10.2.10 **EVENT\_TYPE**

1 EVENT\_TYPE is a derived type with private components. It is an extensible type with no type parameters. Each nonallocatable component is fully default-initialized.

2 A scalar variable of type EVENT\_TYPE is an event variable. The value of an event variable includes its event count, which is updated by execution of a sequence of EVENT POST or EVENT WAIT statements. The effect of each change is as if the intrinsic subroutine ATOMIC\_ADD were executed with a variable that stores the event count as its ATOM argument. A coarray that is of type EVENT\_TYPE may be referenced or defined during execution of a segment that is unordered relative to the execution of another segment in which that coarray is defined. The event count is of type integer with kind ATOMIC\_INT\_KIND from the intrinsic module ISO\_FORTRAN\_ENV. The initial value of the event count of an event variable is zero.

C1603 A named variable with declared type EVENT\_TYPE shall be a coarray. A named variable with a noncoarray subcomponent of type EVENT\_TYPE shall be a coarray.
C1604 An event variable shall not appear in a variable definition context except as the event-variable in an EVENT POST or EVENT WAIT statement, as an allocate-object, or as an actual argument in a reference to a procedure with an explicit interface if the corresponding dummy argument has INTENT (INOUT).

C1605 A variable with a nonpointer subobject of type EVENT_TYPE shall not appear in a variable definition context except as an allocate-object in an ALLOCATE statement without a SOURCE= specifier, as an allocate-object in a DEALLOCATE statement, or as an actual argument in a reference to a procedure with an explicit interface if the corresponding dummy argument has INTENT (INOUT).

NOTE 16.32
The restrictions against changing an event variable except via EVENT POST and EVENT WAIT statements ensure the integrity of its value and facilitate efficient implementation, particularly when special synchronization is needed for correct event handling.

NOTE 16.33
Updates to variables via atomic subroutines are coherent but not necessarily consistent, so a processor might have to use extra synchronization to obtain the consistency required for the segments ordered by EVENT POST and EVENT WAIT statements.

16.10.2.11 FILE_STORAGE_SIZE

1 The value of the default integer scalar constant FILE_STORAGE_SIZE is the size expressed in bits of the file storage unit (12.3.5).

16.10.2.12 INITIAL_TEAM

1 The value of the default integer scalar constant INITIAL_TEAM identifies the initial team when it is used as the LEVEL argument to GET_TEAM.

16.10.2.13 INPUT_UNIT

1 The value of the default integer scalar constant INPUT_UNIT identifies the same processor-dependent external unit preconnected for sequential formatted input as the one identified by an asterisk in a READ statement; this unit is the one used for a READ statement that does not contain an input/output control list (12.6.4.3). The value shall not be −1.

16.10.2.14 INT8, INT16, INT32, and INT64

1 The values of these default integer scalar constants shall be those of the kind type parameters that specify an INTEGER type whose storage size expressed in bits is 8, 16, 32, and 64 respectively. If, for any of these constants, the processor supports more than one kind of that size, it is processor dependent which kind value is provided. If the processor supports no kind of a particular size, that constant shall be equal to −2 if the processor supports a kind with larger size and −1 otherwise.

16.10.2.15 INTEGER_KINDS

1 The values of the elements of the default integer array constant INTEGER_KINDS are the kind values supported by the processor for variables of type integer. The order of the values is processor dependent. The rank of the array is one, its lower bound is one, and its size is the number of integer kinds supported.

16.10.2.16 IOSTAT_END

1 The value of the default integer scalar constant IOSTAT_END is assigned to the variable specified in an IOSTAT= specifier (12.11.5) if an end-of-file condition occurs during execution of an input statement and no error condition occurs. This value shall be negative.
16.10.2.17 IOSTAT_EOR

1 The value of the default integer scalar constant IOSTAT_EOR is assigned to the variable specified in an IOSTAT= specifier (12.11.5) if an end-of-record condition occurs during execution of an input statement and no end-of-file or error condition occurs. This value shall be negative and different from the value of IOSTAT_END.

16.10.2.18 IOSTAT_INQUIRE_INTERNAL_UNIT

1 The value of the default integer scalar constant IOSTAT_INQUIRE_INTERNAL_UNIT is assigned to the variable specified in an IOSTAT= specifier in an INQUIRE statement (12.10) if a file-unit-number identifies an internal unit in that statement.

NOTE 16.34

This can only occur when a defined input/output procedure is called by the processor as the result of executing a parent data transfer statement (12.6.4.8.3) for an internal unit.

16.10.2.19 LOCK_TYPE

1 LOCK_TYPE is a derived type with private components; no component is allocatable or a pointer. It is an extensible type with no type parameters. Therefore it does not have the BIND attribute and is not a sequence type. All components have default initialization.

2 A scalar variable of type LOCK_TYPE is a lock variable. A lock variable can have one of two states: locked and unlocked. The unlocked state is represented by the one value that is the default value of a LOCK_TYPE variable; this is the value specified by the structure constructor LOCK_TYPE ( ). The locked state is represented by all other values. The value of a lock variable can be changed with the LOCK and UNLOCK statements (11.6.10).

C1606 A named variable with declared type LOCK_TYPE shall be a coarray. A named variable with a noncoarray subcomponent of type LOCK_TYPE shall be a coarray.

C1607 A lock variable shall not appear in a variable definition context except as the lock-variable in a LOCK or UNLOCK statement, as an allocate-object, or as an actual argument in a reference to a procedure with an explicit interface where the corresponding dummy argument has INTENT (INOUT).

C1608 A variable with a subobject of type LOCK_TYPE shall not appear in a variable definition context except as an allocate-object or as an actual argument in a reference to a procedure with an explicit interface where the corresponding dummy argument has INTENT (INOUT).

NOTE 16.35

The restrictions against changing a lock variable except via the LOCK and UNLOCK statements ensure the integrity of its value and facilitate efficient implementation, particularly when special synchronization is needed for correct lock operation.

16.10.2.20 LOGICAL_KINDS

1 The values of the elements of the default integer array constant LOGICAL_KINDS are the kind values supported by the processor for variables of type logical. The order of the values is processor dependent. The rank of the array is one, its lower bound is one, and its size is the number of logical kinds supported.

16.10.2.21 NUMERIC_STORAGE_SIZE

1 The value of the default integer scalar constant NUMERIC_STORAGE_SIZE is the size expressed in bits of the numeric storage unit (19.5.3.2).
16.10.2.22 OUTPUT_UNIT

1 The value of the default integer scalar constant OUTPUT_UNIT identifies the same processor-dependent external unit preconnected for sequential formatted output as the one identified by an asterisk in a WRITE statement (12.6.4.3). The value shall not be \(-1\).

16.10.2.23 PARENT_TEAM

1 The value of the default integer scalar constant PARENT_TEAM identifies the parent team when it is used as the LEVEL argument to GET_TEAM.

16.10.2.24 REAL_KINDS

1 The values of the elements of the default integer array constant REAL_KINDS are the kind values supported by the processor for variables of type real. The order of the values is processor dependent. The rank of the array is one, its lower bound is one, and its size is the number of real kinds supported.

16.10.2.25 REAL32, REAL64, and REAL128

1 The values of these default integer scalar named constants shall be those of the kind type parameters that specify a REAL type whose storage size expressed in bits is 32, 64, and 128 respectively. If, for any of these constants, the processor supports more than one kind of that size, it is processor dependent which kind value is provided. If the processor supports no kind of a particular size, that constant shall be equal to \(-2\) if the processor supports kinds of a larger size and \(-1\) otherwise.

16.10.2.26 STAT_FAILED_IMAGE

1 If the processor has the ability to detect that an image has failed, the value of the default integer scalar constant STAT_FAILED_IMAGE is positive; otherwise, the value of STAT_FAILED_IMAGE is negative. If an image involved in execution of an image control statement, a reference to a coindexed object, or execution of a collective or atomic subroutine has failed, and no other error condition occurs, the value of STAT_FAILED_IMAGE is assigned to the variable specified in a STAT= specifier in the execution of an image control statement or reference to a coindexed object, or to the STAT argument in an invocation of a collective or atomic subroutine.

16.10.2.27 STAT_LOCKED

1 The value of the default integer scalar constant STAT_LOCKED is assigned to the variable specified in a STAT= specifier (11.6.11) of a LOCK statement if the lock variable is locked by the executing image.

16.10.2.28 STAT_LOCKED_OTHER_IMAGE

1 The value of the default integer scalar constant STAT_LOCKED_OTHER_IMAGE is assigned to the variable specified in a STAT= specifier (11.6.11) of an UNLOCK statement if the lock variable is locked by another image.

16.10.2.29 STAT_STOPPED_IMAGE

1 The value of the default integer scalar constant STAT_STOPPED_IMAGE is assigned to the variable specified in a STAT= specifier (9.7.4, 11.6.11), if execution of the statement with that specifier requires synchronization with an image that has initiated normal termination. It is assigned to a STAT argument in a reference to a collective subroutine if any image of the current team has initiated normal termination. This value shall be positive.

16.10.2.30 STAT_UNLOCKED

1 The value of the default integer scalar constant STAT_UNLOCKED is assigned to the variable specified in a STAT= specifier (11.6.11) of an UNLOCK statement if the lock variable is unlocked.
16.10.2.31 STAT_UNLOCKED_FAILED_IMAGE

The value of the default integer scalar constant STAT_UNLOCKED_FAILED_IMAGE is assigned to the variable specified in a STAT= specifier (11.6.11) of a LOCK statement if the lock variable is unlocked because of the failure of the image that locked it.

16.10.2.32 TEAM_TYPE

TEAM_TYPE is a derived type with private components; no component is a pointer. It is an extensible type with no type parameters. Therefore it does not have the BIND attribute and is not a sequence type. Each component is fully default-initialized.

A scalar variable of type TEAM_TYPE is a team variable, and can identify a team. The default initial value of a team variable does not identify any team.

16.10.2.33 Uniqueness of named constant values

The values of the named constants IOSTAT_INQUIRE_INTERNAL_UNIT, STAT_FAILED_IMAGE, STAT_LOCKED, STAT_LOCKED_OTHER_IMAGE, STAT_STOPPED_IMAGE, STAT_UNLOCKED, and STAT_UNLOCKED_FAILED_IMAGE shall be distinct.
17 Exceptions and IEEE arithmetic

17.1 Overview of IEEE arithmetic support

1 The intrinsic modules IEEE_EXCEPTIONS, IEEE_ARITHMETIC, and IEEE_FEATURES provide support for the facilities defined by ISO/IEC/IEEE 60559:2011*. Whether the modules are provided is processor dependent. If the module IEEE_FEATURES is provided, which of the named constants defined in this document are included is processor dependent. The module IEEE_ARITHMETIC behaves as if it contained a USE statement for IEEE_EXCEPTIONS; everything that is public in IEEE_EXCEPTIONS is public in IEEE_ARITHMETIC.

NOTE 17.1

The types and procedures defined in these modules are not themselves intrinsic.

2 If IEEE_EXCEPTIONS or IEEE_ARITHMETIC is accessible in a scoping unit, the exceptions IEEE_OVERFLOW and IEEE_DIVIDE_BY_ZERO are supported in the scoping unit for all kinds of real and complex IEEE floating-point data. Which other exceptions are supported can be determined by the inquiry function IEEE_SUPPORT_FLAG (17.11.51); whether control of halting is supported can be determined by the inquiry function IEEE_SUPPORT_HALTING. The extent of support of the other exceptions may be influenced by the accessibility of the named constants IEEE_INEXACT_FLAG, IEEE_INVALID_FLAG, and IEEE_UNDERFLOW_FLAG of the module IEEE_FEATURES. If a scoping unit has access to IEEE_UNDERFLOW_FLAG of IEEE_FEATURES, within the scoping unit the processor shall support underflow and return true from IEEE_SUPPORT_FLAG (IEEE_UNDERFLOW, X) for at least one kind of real. Similarly, if IEEE_INEXACT_FLAG or IEEE_INVALID_FLAG is accessible, within the scoping unit the processor shall support the exception and return true from the corresponding inquiry function for at least one kind of real. If IEEE_HALTING is accessible, within the scoping unit the processor shall support control of halting and return true from IEEE_SUPPORT_HALTING (FLAG) for the flag.

NOTE 17.2

IEEE_INVALID is not required to be supported whenever IEEE_EXCEPTIONS is accessed. This is to allow a processor whose arithmetic does not conform to ISO/IEC/IEEE 60559:2011 to provide support for overflow and divide_by_zero. On a processor which does support ISO/IEC/IEEE 60559:2011, invalid is an equally serious condition.

3 If a scoping unit does not access IEEE_FEATURES, IEEE_EXCEPTIONS, or IEEE_ARITHMETIC, the level of support is processor dependent, and need not include support for any exceptions. If a flag is signaling on entry to such a scoping unit, the processor ensures that it is signaling on exit. If a flag is quiet on entry to such a scoping unit, whether it is signaling on exit is processor dependent.

4 Additional ISO/IEC/IEEE 60559:2011 facilities are available from the module IEEE_ARITHMETIC. The extent of support may be influenced by the accessibility of the named constants of the module IEEE_FEATURES. If a scoping unit has access to IEEE_DATATYPE of IEEE_FEATURES, within the scoping unit the processor shall support IEEE arithmetic and return true from IEEE_SUPPORT_DATATYPE (X) (17.11.48) for at least one kind of real. Similarly, if IEEE_DENORMAL, IEEE_DIVIDE, IEEE_INF, IEEE_NAN, IEEE_ROUNDING, IEEE_SQRT, or IEEE_SUBNORMAL is accessible, within the scoping unit the processor shall support the feature and return true from the corresponding inquiry function for at least one kind of real. In the case of IEEE_ROUNDING, it shall return true for the rounding modes IEEE_NEAREST, IEEE_TO_ZERO, IEEE_UP, and IEEE_DOWN; support for IEEE_AWAY is also required if there is at least one kind of real X for

* Because ISO/IEC/IEEE 60559:2011 was originally an IEEE standard, its facilities are widely known as “IEEE arithmetic”, and this terminology is used by this document.
which \texttt{IEEE\_SUPPORT\_DATATYPE} (X) is true and \texttt{RADIX} (X) is equal to ten. Note that the effect of \texttt{IEEE\_DENORMAL} is the same as that of \texttt{IEEE\_SUBNORMAL}.

Execution might be slowed on some processors by the support of some features. If \texttt{IEEE\_EXCEPTIONS} or \texttt{IEEE\_ARITHMETIC} is accessed but \texttt{IEEE\_FEATURES} is not accessed, the supported subset of features is processor dependent. The processor’s fullest support is provided when all of \texttt{IEEE\_FEATURES} is accessed as in

\begin{verbatim}
USE, INTRINSIC :: IEEE\_ARITHMETIC; USE, INTRINSIC :: IEEE\_FEATURES
\end{verbatim}

but execution might then be slowed by the presence of a feature that is not needed. In all cases, the extent of support can be determined by the inquiry functions.

### 17.2 Derived types, constants, and operators defined in the modules

The modules \texttt{IEEE\_EXCEPTIONS}, \texttt{IEEE\_ARITHMETIC}, and \texttt{IEEE\_FEATURES} define five derived types, whose components are all private. No direct component of any of these types is allocatable or a pointer.

The module \texttt{IEEE\_EXCEPTIONS} defines the following types.

- \texttt{IEEE\_FLAG\_TYPE} is for identifying a particular exception flag. Its only possible values are those of named constants defined in the module: \texttt{IEEE\_INVALID}, \texttt{IEEE\_OVERFLOW}, \texttt{IEEE\_DIVIDE\_BY\_ZERO}, \texttt{IEEE\_UNDERFLOW}, and \texttt{IEEE\_INEXACT}. The module also defines the array named constants \texttt{IEEE\_USUAL} = \texttt{[IEEE\_OVERFLOW, IEEE\_DIVIDE\_BY\_ZERO, IEEE\_INVALID]} and \texttt{IEEE\_ALL} = \texttt{[IEEE\_USUAL, IEEE\_UNDERFLOW, IEEE\_INEXACT]}.

- \texttt{IEEE\_MODES\_TYPE} is for representing the floating-point modes.

- \texttt{IEEE\_STATUS\_TYPE} is for representing the floating-point status.

The module \texttt{IEEE\_ARITHMETIC} defines the following types, constants, and operators.

- The type \texttt{IEEE\_CLASS\_TYPE}, for identifying a class of floating-point values. Its only possible values are those of named constants defined in the module: \texttt{IEEE\_SIGNALING\_NAN}, \texttt{IEEE\_QUIET\_NAN}, \texttt{IEEE\_NEGATIVE\_INF}, \texttt{IEEE\_NEGATIVE\_NORMAL}, \texttt{IEEE\_NEGATIVE\_DENORMAL}, \texttt{IEEE\_NEGATIVE\_ZERO}, \texttt{IEEE\_POSITIVE\_ZERO}, \texttt{IEEE\_POSITIVE\_NORMAL}, \texttt{IEEE\_POSITIVE\_SUBNORMAL}, \texttt{IEEE\_POSITIVE\_INF}, and \texttt{IEEE\_OTHER\_VALUE}. The named constants \texttt{IEEE\_NEGATIVE\_DENORMAL} and \texttt{IEEE\_POSITIVE\_DENORMAL} are defined with the same value as \texttt{IEEE\_NEGATIVE\_SUBNORMAL} and \texttt{IEEE\_POSITIVE\_SUBNORMAL} respectively.

- The type \texttt{IEEE\_ROUND\_TYPE}, for identifying a particular rounding mode. Its only possible values are those of named constants defined in the module: \texttt{IEEE\_NEAREST}, \texttt{IEEE\_TO\_ZERO}, \texttt{IEEE\_UP}, \texttt{IEEE\_DOWN}, and \texttt{IEEE\_AWAY} for the rounding modes specified in ISO/IEC/IEEE 60559:2011, and \texttt{IEEE\_OTHER} for any other mode.

- The pure elemental operator \texttt{==} for two values of one of these types to return true if the values are the same and false otherwise.

- The pure elemental operator \texttt{/=} for two values of one of these types to return true if the values differ and false otherwise.

The module \texttt{IEEE\_FEATURES} defines the type \texttt{IEEE\_FEATURES\_TYPE}, for expressing the need for particular ISO/IEC/IEEE 60559:2011 features. Its only possible values are those of named constants defined in the module: \texttt{IEEE\_DATATYPE}, \texttt{IEEE\_DENORMAL}, \texttt{IEEE\_DIVIDE}, \texttt{IEEE\_HALTING}, \texttt{IEEE\_INEXACT\_FLAG}, \texttt{IEEE\_INF}, \texttt{IEEE\_INVALID\_FLAG}, \texttt{IEEE\_NAN}, \texttt{IEEE\_ROUNDING}, \texttt{IEEE\_SQRT}, \texttt{IEEE\_SUBNORMAL}, and \texttt{IEEE\_UNDERFLOW\_FLAG}.

### 17.3 The exceptions

The exceptions are the following.
• IEEE_OVERFLOW occurs in an intrinsic real addition, subtraction, multiplication, division, or conversion by the intrinsic function REAL, as specified by ISO/IEC/IEEE 60559:2011 if IEEE_SUPPORT_DATATYPE is true for the operands of the operation or conversion, and as determined by the processor otherwise. It occurs in an intrinsic real exponentiation as determined by the processor. It occurs in a complex operation, or conversion by the intrinsic function CMPLX, if it is caused by the calculation of the real or imaginary part of the result.

• IEEE_DIVIDE_BY_ZERO occurs in a real division as specified by ISO/IEC/IEEE 60559:2011 if IEEE_SUPPORT_DATATYPE is true for the operands of the division, and as determined by the processor otherwise. It is processor-dependent whether it occurs in a real exponentiation with a negative exponent. It occurs in a complex division if it is caused by the calculation of the real or imaginary part of the result.

• IEEE_INVALID occurs when a real or complex operation or assignment is invalid; possible examples are SQRT (X) when X is real and has a nonzero negative value, and conversion to an integer (by assignment, an intrinsic procedure, or a procedure defined in an intrinsic module) when the result is too large to be representable. IEEE_INVALID occurs for numeric relational intrinsic operations as specified below.

• IEEE_UNDERFLOW occurs when the result for an intrinsic real operation or assignment has an absolute value less than a processor-dependent limit and loss of accuracy is detected, or the real or imaginary part of the result for an intrinsic complex operation or assignment has an absolute value less than a processor-dependent limit and loss of accuracy is detected.

• IEEE_INEXACT occurs when the result of a real or complex operation or assignment is not exact.

2 Each exception has a flag whose value is either quiet or signaling. The value can be determined by the subroutine IEEE_GET_FLAG. Its initial value is quiet and it signals when the associated exception occurs. Its status can also be changed by the subroutine IEEE_SET_FLAGS or the subroutine IEEE_SET_STATUS. Once signaling within a procedure, it remains signaling unless set quiet by an invocation of the subroutine IEEE_SET_FLAG or the subroutine IEEE_SET_STATUS.

3 If a flag is signaling on entry to a procedure other than IEEE_GET_FLAG or IEEE_GET_STATUS, the processor will set it to quiet on entry and restore it to signaling on return. If a flag signals during execution of a procedure, the processor shall not set it to quiet on return.

4 Evaluation of a specification expression might cause an exception to signal.

5 In a scoping unit that has access to IEEE_EXCEPTIONS or IEEE_ARITHMETIC, if an intrinsic procedure or a procedure defined in an intrinsic module executes normally, the values of the flags IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, and IEEE_INVALID shall be as on entry to the procedure, even if one or more of them signals during the calculation. If a real or complex result is too large for the procedure to handle, IEEE_OVERFLOW may signal. If a real or complex result is a NaN because of an invalid operation (for example, LOG (-1.0)), IEEE_INVALID may signal. Similar rules apply to format processing and to intrinsic operations: no signaling flag shall be set quiet and no quiet flag shall be set signaling because of an intermediate calculation that does not affect the result.

6 In a scoping unit that has access to IEEE_EXCEPTIONS or IEEE_ARITHMETIC, if x₁ and x₂ are numeric entities, the type of x₁ + x₂ is real, and IEEE_SUPPORT_NAN (x₁ + x₂) is true, the relational intrinsic operation x₁ rel-op x₂ shall signal IEEE_INVALID as specified for the conditional predicate of ISO/IEC/IEEE 60559:2011 corresponding to rel-op indicated by Table 17.1. If the types or kind type parameters of x₁ or x₂ differ, the conversions (10.1.5.5.1) might signal exceptions instead of or in addition to an IEEE_INVALID exception signaled by the comparison.

NOTE 17.3

Each comparison predicate defined by ISO/IEC/IEEE 60559:2011 is either unordered signaling or unordered quiet. An unordered signaling predicate signals an invalid operation exception if and only if one of the values being compared is a NaN. An unordered quiet predicate signals an invalid operation exception if and only if one of the values being compared is a signaling NaN. The comparison predicates do not signal any other exceptions.
Table 17.1: IEEE relational operator correspondence

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>.LT. or &lt;</td>
<td>compareSignalingLess</td>
</tr>
<tr>
<td>.LE. or &lt;=</td>
<td>compareSignalingLessEqual</td>
</tr>
<tr>
<td>.GE. or &gt;=</td>
<td>compareSignalingGreaterEqual</td>
</tr>
<tr>
<td>.GT. or &gt;</td>
<td>compareSignalingGreater</td>
</tr>
<tr>
<td>.EQ. or ==</td>
<td>compareQuietEqual</td>
</tr>
<tr>
<td>.NE. or /=</td>
<td>compareQuietNotEqual</td>
</tr>
</tbody>
</table>

7 In a scoping unit that has access to IEEE_EXCEPTIONS or IEEE_ARITHMETIC, if \( x_1 \) or \( x_2 \) are numeric entities, the type of \( x_1 + x_2 \) is complex, and IEEE_SUPPORT_NAN (REAL \( (x_1 + x_2) \)) is true, the relational operation may signal IEEE_INVALID if the value of the real or imaginary part of either operand is a signaling NaN. If any conversions are done before the values are compared, those conversions might signal exceptions instead of or in addition to an IEEE_INVALID exception signaled by the comparison.

8 In a sequence of statements that has no invocations of IEEE_GET_FLAG, IEEE_SET_FLAG, IEEE_GET_STATUS, IEEE_SET_HALTING_MODE, or IEEE_SET_STATUS, 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 is processor dependent. For example, when \( Y \) has the value zero, whether the code

\[
X = 1.0 / Y \\
X = 3.0
\]

signals IEEE_DIVIDE_BY_ZERO is processor dependent. Another example is the following:

REAL, PARAMETER :: X=0.0, Y=6.0  
IF (1.0/X == Y) PRINT *, 'Hello world'

where the processor is permitted to discard the IF statement because the logical expression can never be true and no value of a variable depends on it.

9 An exception shall not signal if this could arise only during execution of an operation beyond those required or permitted by the standard. For example, the statement

\[
\text{IF} ~(F(X) > 0.0)~ Y = 1.0 / Z
\]

shall not signal IEEE_DIVIDE_BY_ZERO when both \( F(X) \) and \( Z \) are zero and the statement

\[
\text{WHERE} (A > 0.0)~ A = 1.0 / A
\]

shall not signal IEEE_DIVIDE_BY_ZERO. On the other hand, when \( X \) has the value 1.0 and \( Y \) has the value 0.0, the expression

\[
X>0.00001 \text{ OR } X/Y>0.00001
\]

is permitted to cause the signaling of IEEE_DIVIDE_BY_ZERO.

10 The processor need not support IEEE_INVALID, IEEE_UNDERFLOW, and IEEE_INEXACT. If an exception is not supported, its flag is always quiet. The inquiry function IEEE_SUPPORT_FLAG can be used to inquire whether a particular flag is supported.

17.4 The rounding modes

ISO/IEC/IEEE 60559:2011 specifies a binary rounding mode that affects floating-point arithmetic with radix two, and a decimal rounding mode that affects floating-point arithmetic with radix ten. Unqualified references
to the rounding mode with respect to a particular arithmetic operation or operands refers to the mode for the
radix of the operation or operands, and other unqualified references to the rounding mode refers to both binary
and decimal rounding modes.

ISO/IEC/IEEE 60559:2011 specifies five possible modes for rounding:

- IEEE_NEAREST rounds the exact result to the nearest representable value;
- IEEE_TO_ZERO rounds the exact result towards zero to the next representable value;
- IEEE_UP rounds the exact result towards $+\infty$ to the next representable value;
- IEEE_DOWN rounds the exact result towards $-\infty$ to the next representable value;
- IEEE_AWAY rounds the exact result away from zero to the next representable value; ISO/IEC/IEEE
  60559:2011 requires this mode for decimal floating-point, but it is optional for binary floating-point.

The subroutine IEEE_GET_ROUNDING_MODE can be used to get the rounding modes. The initial rounding
modes are processor dependent.

If the processor supports the alteration of the rounding modes during execution, the subroutine IEEE_SET_-ROUNDING_MODE can be used to alter them. The inquiry function IEEE_SUPPORT_ROUNDING can be
used to inquire whether this facility is available for a particular mode. The inquiry function IEEE_SUPPORT_-IO can be used to inquire whether rounding for base conversion in formatted input/output (12.5.6.16, 12.6.2.13,
13.7.2.3.8) is as specified in ISO/IEC/IEEE 60559:2011.

In a procedure other than IEEE_SET_ROUNDING_MODE or IEEE_SET_STATUS, the processor shall not
change the rounding modes on entry, and on return shall ensure that the rounding modes are the same as they
were on entry.

NOTE 17.4

Within a program, all literal constants that have the same form have the same value (7.1.4). Therefore, the
value of a literal constant is not affected by the rounding modes.

17.5 Underflow mode

Some processors allow control during program execution of whether underflow produces a subnormal number in
conformance with ISO/IEC/IEEE 60559:2011 (gradual underflow) or produces zero instead (abrupt underflow).
On some processors, floating-point performance is typically better in abrupt underflow mode than in gradual
underflow mode.

Control over the underflow mode is exercised by invocation of IEEE_SET_UNDERFLOW_MODE. The sub-
routine IEEE_GET_UNDERFLOW_MODE can be used to get the underflow mode. The inquiry function
IEEE_SUPPORT_UNDERFLOW_CONTROL can be used to inquire whether this facility is available. The
initial underflow mode is processor dependent. In a procedure other than IEEE_SET_UNDERFLOW_MODE
or IEEE_SET_STATUS, the processor shall not change the underflow mode on entry, and on return shall ensure
that the underflow mode is the same as it was on entry.

The underflow mode affects only floating-point calculations whose type is that of an X for which IEEE_SUP-
PORT_UNDERFLOW_CONTROL returns true.

17.6 Halting

Some processors allow control during program execution of whether to abort or continue execution after an
exception. Such control is exercised by invocation of the subroutine IEEE_SET_HALTING_MODE. Halting
is not precise and may occur any time after the exception has occurred. The inquiry function IEEE_SUP-
PORT_HALTING can be used to inquire whether this facility is available. The initial halting mode is processor
dependent. In a procedure other than IEEE_SET_HALTING_MODE or IEEE_SET_STATUS, the processor
shall not change the halting mode on entry, and on return shall ensure that the halting mode is the same as it
was on entry.

17.7 The floating-point modes and status

The values of the rounding modes, underflow mode, and halting mode are collectively called the floating-point
modes. The values of all the supported flags for exceptions and the floating-point modes are collectively called the
floating-point status. The floating-point modes can be stored in a scalar variable of type IEEE_MODES_TYPE
with the subroutine IEEE_GET_MODES and restored with the subroutine IEEE_SET_MODES. The floating-
point status can be stored in a scalar variable of type IEEE_STATUS_TYPE with the subroutine IEEE_GET_-STATUS and restored with the subroutine IEEE_SET_STATUS. There are no facilities for finding the values of
particular flags represented by such a variable.

NOTE 17.5

Each image has its own floating-point status (5.3.4).

NOTE 17.6

Some processors hold all these flags and modes in one or two status registers that can be obtained and
set as a whole faster than all individual flags and modes can be obtained and set. These procedures are
provided to exploit this feature.

NOTE 17.7

The processor is required to ensure that a call to a Fortran procedure does not change the floating-point
status other than by setting exception flags to signaling.

17.8 Exceptional values

ISO/IEC/IEEE 60559:2011 specifies the following exceptional floating-point values.

• Subnormal values have very small absolute values and reduced precision.
• Infinite values (+infinity and −infinity) are created by overflow or division by zero.
• Not-a-Number (NaN) values are undefined values or values created by an invalid operation.

A value that does not fall into the above classes is called a normal number.

The functions IEEE_IS_FINITE, IEEE_IS_NAN, IEEE_IS_NEGATIVE, and IEEE_IS_NORMAL are
provided to test whether a value is finite, NaN, negative, or normal. The function IEEE_VALUE is provided to
generate an IEEE number of any class, including an infinity or a NaN. The inquiry functions IEEE_SUPPORT_-SUBNORMAL, IEEE_SUPPORT_INF, and IEEE_SUPPORT_NAN are provided to determine whether these
facilities are available for a particular kind of real.

17.9 IEEE arithmetic

The inquiry function IEEE_SUPPORT_DATATYPE can be used to inquire whether IEEE arithmetic is sup-
ported for a particular kind of real. Complete conformance with ISO/IEC/IEEE 60559:2011 is not required, but

• the normal numbers shall be exactly those of an ISO/IEC/IEEE 60559:2011 floating-point format,
• for at least one rounding mode, the intrinsic operations of addition, subtraction and multiplication shall
  conform whenever the operands and result specified by ISO/IEC/IEEE 60559:2011 are normal numbers,
• the IEEE function abs shall be provided by the intrinsic function ABS,
• the IEEE operation rem shall be provided by the function IEEE_REM, and
• the IEEE functions copysign, scalb, logb, nextafter, and unordered shall be provided by the functions
  IEEE_COPY_SIGN, IEEE_SCALB, IEEE_LOGB, IEEE_NEXT_AFTER, and IEEE_UNORDERED, respectively,
for that kind of real.

2 The inquiry function IEEE_SUPPORT_NAN is provided to inquire whether the processor supports IEEE NaNs.
Where these are supported, the result of the intrinsic operations +, −, and *, and the functions IEEE_REM
and IEEE_RINT from the intrinsic module IEEE_ARITHMETIC, shall conform to ISO/IEC/IEEE 60559:2011
when the result is an IEEE NaN.

3 The inquiry function IEEE_SUPPORT_INF is provided to inquire whether the processor supports IEEE infinit-
ies. Where these are supported, the result of the intrinsic operations +, −, and *, and the functions IEEE_REM
and IEEE_RINT from the intrinsic module IEEE_ARITHMETIC, shall conform to ISO/IEC/IEEE 60559:2011
when exactly one operand or the result specified by ISO/IEC/IEEE 60559:2011 is an IEEE infinity.

4 The inquiry function IEEE_SUPPORT_SUBNORMAL is provided to inquire whether the processor supports
subnormal numbers. Where these are supported, the result of the intrinsic operations +, −, and *, and the
functions IEEE_REM and IEEE_RINT from the intrinsic module IEEE_ARITHMETIC, shall conform to
ISO/IEC/IEEE 60559:2011 when the result specified by ISO/IEC/IEEE 60559:2011 is subnormal, or any op-
erand is subnormal and either the result is not an IEEE infinity or IEEE_SUPPORT_INF is true.

5 The inquiry function IEEE_SUPPORT_DIVIDE is provided to inquire whether, on kinds of real for which
IEEE_SUPPORT_DATATYPE returns true, the intrinsic division operation conforms to ISO/IEC/IEEE
60559:2011 when both operands and the result specified by ISO/IEC/IEEE 60559:2011 are normal numbers.
If IEEE_SUPPORT_NAN is also true for a particular kind of real, the intrinsic division operation on that kind
conforms to ISO/IEC/IEEE 60559:2011 when the result specified by ISO/IEC/IEEE 60559:2011 is a NaN. If
IEEE_SUPPORT_INF is also true for a particular kind of real, the intrinsic division operation on that kind
conforms to ISO/IEC/IEEE 60559:2011 when one operand or the result specified by ISO/IEC/IEEE 60559:2011 is
an IEEE infinity. If IEEE_SUPPORT_SUBNORMAL is also true for a particular kind of real, the intrinsic divi-
sion operation on that kind conforms to ISO/IEC/IEEE 60559:2011 when the result specified by ISO/IEC/IEEE
60559:2011 is subnormal, or when any operand is subnormal and either the result specified by ISO/IEC/IEEE
60559:2011 is not an infinity or IEEE_SUPPORT_INF is true.

6 ISO/IEC/IEEE 60559:2011 specifies a square root function that returns negative real zero for the square root of
negative real zero and has certain accuracy requirements. The inquiry function IEEE_SUPPORT_SQRT can
be used to inquire whether the intrinsic function SQRT conforms to ISO/IEC/IEEE 60559:2011 for a particular
kind of real. If IEEE_SUPPORT_NAN is also true for a particular kind of real, the intrinsic function SQRT
on that kind conforms to ISO/IEC/IEEE 60559:2011 when the result specified by ISO/IEC/IEEE 60559:2011 is a NaN. If
IEEE_SUPPORT_INF is also true for a particular kind of real, the intrinsic function SQRT on
that kind conforms to ISO/IEC/IEEE 60559:2011 when the result specified by ISO/IEC/IEEE 60559:2011 is an
IEEE infinity. If IEEE_SUPPORT_SUBNORMAL is also true for a particular kind of real, the intrinsic function
SQRT on that kind conforms to ISO/IEC/IEEE 60559:2011 when the argument is subnormal.

7 The inquiry function IEEE_SUPPORT_STANDARD is provided to inquire whether the processor supports all
the ISO/IEC/IEEE 60559:2011 facilities defined in this document for a particular kind of real.

17.10 Summary of the procedures

1 For all of the procedures defined in the modules, the arguments shown are the names that shall be used for
argument keywords if the keyword form is used for the actual arguments.

2 A procedure classified in 17.10 as an inquiry function depends on the properties of one or more of its arguments
instead of their values; in fact, these argument values may be undefined. Unless the description of one of these
inquiry functions states otherwise, these arguments are permitted to be unallocated allocatable variables or
pointers that are undefined or disassociated. A procedure that is classified as a transformational function is
In the Class column of Tables 17.2 and 17.3,

E indicates that the procedure is an elemental function,
ES indicates that the procedure is an elemental subroutine,
I indicates that the procedure is an inquiry function,
PS indicates that the procedure is a pure subroutine,
S indicates that the procedure is an impure subroutine, and
T indicates that the procedure is in a transformational function.

Table 17.2: IEEE_ARITHMETIC module procedure summary

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Arguments</th>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE_CLASS</td>
<td>(X)</td>
<td>E</td>
<td>Classify number.</td>
</tr>
<tr>
<td>IEEE_COPY_SIGN</td>
<td>(X, Y)</td>
<td>E</td>
<td>Copy sign.</td>
</tr>
<tr>
<td>IEEE_FMA</td>
<td>(A, B, C)</td>
<td>E</td>
<td>Fused multiply-add operation.</td>
</tr>
<tr>
<td>IEEE_GET_ROUNDING_MODE</td>
<td>(ROUND_VALUE [, RADIX])</td>
<td>S</td>
<td>Get rounding mode.</td>
</tr>
<tr>
<td>IEEE_GET_UNDERFLOW_MODE</td>
<td>(GRADUAL)</td>
<td>S</td>
<td>Get underflow mode.</td>
</tr>
<tr>
<td>IEEE_INT</td>
<td>(A, ROUND [, KIND])</td>
<td>E</td>
<td>Conversion to integer type.</td>
</tr>
<tr>
<td>IEEE_ISFINITE</td>
<td>(X)</td>
<td>E</td>
<td>Whether a value is finite.</td>
</tr>
<tr>
<td>IEEE_IS_NAN</td>
<td>(X)</td>
<td>E</td>
<td>Whether a value is an IEEE NaN.</td>
</tr>
<tr>
<td>IEEE_IS_NEGATIVE</td>
<td>(X)</td>
<td>E</td>
<td>Whether a value is negative.</td>
</tr>
<tr>
<td>IEEE_IS_NORMAL</td>
<td>(X)</td>
<td>E</td>
<td>Whether a value is a normal number.</td>
</tr>
<tr>
<td>IEEE_LOGB</td>
<td>(X)</td>
<td>E</td>
<td>Exponent.</td>
</tr>
<tr>
<td>IEEE_MAX_NUM</td>
<td>(X, Y)</td>
<td>E</td>
<td>Maximum numeric value.</td>
</tr>
<tr>
<td>IEEE_MAX_NUM_MAG</td>
<td>(X, Y)</td>
<td>E</td>
<td>Maximum magnitude numeric value.</td>
</tr>
<tr>
<td>IEEE_MIN_NUM</td>
<td>(X, Y)</td>
<td>E</td>
<td>Minimum numeric value.</td>
</tr>
<tr>
<td>IEEE_MIN_NUM_MAG</td>
<td>(X, Y)</td>
<td>E</td>
<td>Minimum magnitude numeric value.</td>
</tr>
<tr>
<td>IEEE_NEXT_AFTER</td>
<td>(X, Y)</td>
<td>E</td>
<td>Adjacent machine number.</td>
</tr>
<tr>
<td>IEEE_NEXT_DOWN</td>
<td>(X)</td>
<td>E</td>
<td>Adjacent lower machine number.</td>
</tr>
<tr>
<td>IEEE_NEXT_UP</td>
<td>(X)</td>
<td>E</td>
<td>Adjacent higher machine number.</td>
</tr>
<tr>
<td>IEEE_QUIET_EQ</td>
<td>(A, B)</td>
<td>E</td>
<td>Quiet compares equal.</td>
</tr>
<tr>
<td>IEEE_QUIET_GE</td>
<td>(A, B)</td>
<td>E</td>
<td>Quiet compares greater than or equal.</td>
</tr>
<tr>
<td>IEEE_QUIET_GT</td>
<td>(A, B)</td>
<td>E</td>
<td>Quiet compares greater than.</td>
</tr>
<tr>
<td>IEEE_QUIET_LT</td>
<td>(A, B)</td>
<td>E</td>
<td>Quiet compares less than or equal.</td>
</tr>
<tr>
<td>IEEE_QUIET_NE</td>
<td>(A, B)</td>
<td>E</td>
<td>Quiet compares not equal.</td>
</tr>
<tr>
<td>IEEE_REAL</td>
<td>(A [, KIND])</td>
<td>E</td>
<td>Conversion to real type.</td>
</tr>
<tr>
<td>IEEE_REM</td>
<td>(X, Y)</td>
<td>E</td>
<td>Exact remainder.</td>
</tr>
<tr>
<td>IEEE_RINT</td>
<td>(X)</td>
<td>E</td>
<td>Round to integer.</td>
</tr>
<tr>
<td>IEEE_SCALB</td>
<td>(X, I)</td>
<td>E</td>
<td>$X \times 2^I$.</td>
</tr>
<tr>
<td>IEEE.Selected_REAL_KIND</td>
<td>([P, R, RADIX])</td>
<td>T</td>
<td>IEEE kind type parameter value.</td>
</tr>
<tr>
<td>IEEE_SET_ROUNDING_MODE</td>
<td>(ROUND_VALUE [, RADIX])</td>
<td>S</td>
<td>Set rounding mode.</td>
</tr>
<tr>
<td>IEEE_SET_UNDERFLOW_MODE</td>
<td>(GRADUAL)</td>
<td>S</td>
<td>Set underflow mode.</td>
</tr>
<tr>
<td>IEEE_SIGNALING_EQ</td>
<td>(A, B)</td>
<td>E</td>
<td>Signaling compares equal.</td>
</tr>
<tr>
<td>IEEE_SIGNALING_GE</td>
<td>(A, B)</td>
<td>E</td>
<td>Signaling compares greater than or equal.</td>
</tr>
<tr>
<td>IEEE_SIGNALING_GT</td>
<td>(A, B)</td>
<td>E</td>
<td>Signaling compares greater than.</td>
</tr>
<tr>
<td>IEEE_SIGNALING_LE</td>
<td>(A, B)</td>
<td>E</td>
<td>Signaling compares less than or equal.</td>
</tr>
<tr>
<td>IEEE_SIGNALING_LT</td>
<td>(A, B)</td>
<td>E</td>
<td>Signaling compares less than.</td>
</tr>
<tr>
<td>IEEE_SIGNALING_NE</td>
<td>(A, B)</td>
<td>E</td>
<td>Signaling compares not equal.</td>
</tr>
<tr>
<td>IEEE_SIGNBIT</td>
<td>(X)</td>
<td>E</td>
<td>Test sign bit.</td>
</tr>
</tbody>
</table>
Table 17.2: IEEE_ARITHMETIC module procedure summary (cont.)

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Arguments</th>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE_SUPPORT_DATATYPE</td>
<td>([X])</td>
<td>I</td>
<td>Query IEEE arithmetic support.</td>
</tr>
<tr>
<td>IEEE_SUPPORT_DENORMAL</td>
<td>([X])</td>
<td>I</td>
<td>Query subnormal number support.</td>
</tr>
<tr>
<td>IEEE_SUPPORT_DIVIDE</td>
<td>([X])</td>
<td>I</td>
<td>Query IEEE division support.</td>
</tr>
<tr>
<td>IEEE_SUPPORT_INF</td>
<td>([X])</td>
<td>I</td>
<td>Query IEEE infinity support.</td>
</tr>
<tr>
<td>IEEE_SUPPORT_IO</td>
<td>([X])</td>
<td>I</td>
<td>Query IEEE formatting support.</td>
</tr>
<tr>
<td>IEEE_SUPPORT_NAN</td>
<td>([X])</td>
<td>I</td>
<td>Query IEEE NaN support.</td>
</tr>
<tr>
<td>IEEE_SUPPORT_ROUNDING</td>
<td>(ROUND_VALUE [, X])</td>
<td>T</td>
<td>Query IEEE rounding support.</td>
</tr>
<tr>
<td>IEEE_SUPPORT_SQRT</td>
<td>([X])</td>
<td>I</td>
<td>Query IEEE square root support.</td>
</tr>
<tr>
<td>IEEE_SUPPORT_SUBNORMAL</td>
<td>([X])</td>
<td>I</td>
<td>Query subnormal number support.</td>
</tr>
<tr>
<td>IEEE_SUPPORT_STANDARD</td>
<td>([X])</td>
<td>I</td>
<td>Query IEEE standard support.</td>
</tr>
<tr>
<td>IEEE_SUPPORT_UNDERFLOW_CONTROL</td>
<td>([X])</td>
<td>I</td>
<td>Query underflow control support.</td>
</tr>
<tr>
<td>IEEE_UNORDERED</td>
<td>(X, Y)</td>
<td>E</td>
<td>Whether two values are unordered.</td>
</tr>
<tr>
<td>IEEE_VALUE</td>
<td>(X, CLASS)</td>
<td>E</td>
<td>Return number in a class.</td>
</tr>
</tbody>
</table>

Table 17.3: IEEE_EXCEPTIONS module procedure summary

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Arguments</th>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE_GET_FLAG</td>
<td>(FLAG, FLAG_VALUE)</td>
<td>ES</td>
<td>Get an exception flag.</td>
</tr>
<tr>
<td>IEEE_GET_HALTING_MODE</td>
<td>(FLAG, HALTING)</td>
<td>ES</td>
<td>Get a halting mode.</td>
</tr>
<tr>
<td>IEEE_GET_MODES</td>
<td>(MODES)</td>
<td>S</td>
<td>Get floating-point modes.</td>
</tr>
<tr>
<td>IEEE_GET_STATUS</td>
<td>(STATUS_VALUE)</td>
<td>S</td>
<td>Get floating-point status.</td>
</tr>
<tr>
<td>IEEE_SET_FLAG</td>
<td>(FLAG, FLAG_VALUE)</td>
<td>PS</td>
<td>Set an exception flag.</td>
</tr>
<tr>
<td>IEEE_SET_HALTING_MODE</td>
<td>(FLAG, HALTING)</td>
<td>PS</td>
<td>Set a halting mode.</td>
</tr>
<tr>
<td>IEEE_SET_MODES</td>
<td>(MODES)</td>
<td>S</td>
<td>Set floating-point modes.</td>
</tr>
<tr>
<td>IEEE_SET_STATUS</td>
<td>(STATUS_VALUE)</td>
<td>S</td>
<td>Restore floating-point status.</td>
</tr>
<tr>
<td>IEEE_SUPPORT_FLAG</td>
<td>(FLAG [, X])</td>
<td>T</td>
<td>Query exception support.</td>
</tr>
<tr>
<td>IEEE_SUPPORT_HALTING</td>
<td>(FLAG)</td>
<td>T</td>
<td>Query halting mode support.</td>
</tr>
</tbody>
</table>

1 In the intrinsic module IEEE_ARITHMETIC, the elemental functions listed are provided for all reals X and Y.

17.11 Specifications of the procedures

17.11.1 General

1 In the detailed descriptions in 17.11, procedure names are generic and are not specific. All the functions are pure and all the subroutines are impure unless otherwise stated. All dummy arguments have INTENT (IN) if the intent is not stated explicitly. In the examples, it is assumed that the processor supports IEEE arithmetic for default real.

2 For the elemental functions of IEEE_ARITHMETIC that return a floating-point result, if X or Y has a value that is an infinity or a NaN, the result shall be consistent with the general rules in 6.1 and 6.2 of ISO/IEC/IEEE 60559:2011. For example, the result for an infinity shall be constructed as the limiting case of the result with a value of arbitrarily large magnitude, if such a limit exists.

3 A program may contain statements that, if executed, would violate the requirements listed in a Restriction paragraph.

NOTE 17.8

A program can avoid violating those requirements by using IF constructs to check whether particular features are supported. For example,
NOTE 17.8 (cont.)

```
IF (IEEE_SUPPORT_DATATYPE (X)) THEN
  C = IEEE_CLASS (X)
ELSE
  ...
END IF
```

avoids invoking IEEE_CLASS except on a processor which supports that facility.

17.11.2 IEEE_CLASS (X)

1 Description. Classify number.

2 Class. Elemental function.

3 Argument. X shall be of type real.

4 Restriction. IEEE_CLASS (X) shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the value false.

5 Result Characteristics. IEEE_CLASS_TYPE.

6 Result Value. The result value shall be IEEE_SIGNALING_NAN or IEEE_QUIET_NAN if IEEE_SUPPORT_NAN (X) has the value true and the value of X is a signaling or quiet NaN, respectively. The result value shall be IEEE_NEGATIVE_INF or IEEE_POSITIVE_INF if IEEE_SUPPORT_INF (X) has the value true and the value of X is negative or positive infinity, respectively. The result value shall be IEEE_NEGATIVE_SUBNORMAL or IEEE_POSITIVE_SUBNORMAL if IEEE_SUPPORT_SUBNORMAL (X) has the value true and the value of X is negative or positive subnormal value, respectively. The result value shall be IEEE_NEGATIVE_NORMAL, IEEE_NEGATIVE_ZERO, IEEE_POSITIVE_ZERO, or IEEE_POSITIVE_NORMAL if the value of X is negative normal, negative zero, positive zero, or positive normal, respectively. Otherwise, the result value shall be IEEE_OTHER_VALUE.

7 Example. IEEE_CLASS (−1.0) has the value IEEE_NEGATIVE_NORMAL.

NOTE 17.9

The result value IEEE_OTHER_VALUE is useful on systems that are almost IEEE-compatible, but do not implement all of it. For example, if a subnormal value is encountered on a system that does not support them.

17.11.3 IEEE_COPY_SIGN (X, Y)

1 Description. Copy sign.

2 Class. Elemental function.

3 Arguments. The arguments shall be of type real.

4 Restriction. IEEE_COPY_SIGN (X, Y) shall not be invoked if IEEE_SUPPORT_DATATYPE (X) or IEEE_SUPPORT_DATATYPE (Y) has the value false.

5 Result Characteristics. Same as X.

6 Result Value. The result has the absolute value of X with the sign of Y. This is true even for IEEE special values, such as a NaN or an infinity (on processors supporting such values).

7 Example. The value of IEEE_COPY_SIGN (X, 1.0) is ABS (X) even when X is a NaN.
17.11.4  IEEE_FMA (A, B, C)

Description. Fused multiply-add operation.

Class. Elemental function.

Arguments.

A shall be of type real.
B shall be of the same type and kind type parameter as A.
C shall be of the same type and kind type parameter as A.

Restriction. IEEE_FMA (A, B, C) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the value false.

Result Characteristics. Same as A.

Result Value. The result has the value specified by ISO/IEC/IEEE 60559:2011 for the fusedMultiplyAdd operation; that is, when the result is in range, its value is equal to the mathematical value of (A × B) + C rounded to the representation method of A according to the rounding mode. IEEE_OVERFLOW, IEEE_UNDERFLOW, and IEEE_INEXACT shall be signaled according to the final step in the calculation and not by any intermediate calculation.

Example. The value of IEEE_FMA (TINY (0.0), TINY (0.0), 1.0), when the rounding mode is IEEE_NEAREST, is equal to 1.0; only the IEEE_INEXACT exception is signaled.

17.11.5  IEEE_GET_FLAG (FLAG, FLAG_VALUE)

Description. Get an exception flag.

Class. Elemental subroutine.

Arguments.

FLAG shall be of type IEEE_FLAG_TYPE. It specifies the exception flag to be obtained.
FLAG_VALUE shall be of type logical. It is an INTENT (OUT) argument. If the value of FLAG is IEEE_INVALID, IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_UNDERFLOW, or IEEE_INEXACT, FLAG_VALUE is assigned the value true if the corresponding exception flag is signaling and is assigned the value false otherwise.

Example. Following CALL IEEE_GET_FLAG (IEEE_OVERFLOW, FLAG_VALUE), FLAG_VALUE is true if the IEEE_OVERFLOW flag is signaling and is false if it is quiet.

17.11.6  IEEE_GET_HALTING_MODE (FLAG, HALTING)

Description. Get a halting mode.

Class. Elemental subroutine.

Arguments.

FLAG shall be of type IEEE_FLAG_TYPE. It specifies the exception flag. It shall have one of the values IEEE_INVALID, IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_UNDERFLOW, or IEEE_INEXACT.
HALTING shall be of type logical. It is an INTENT (OUT) argument. It is assigned the value true if the exception specified by FLAG will cause halting. Otherwise, it is assigned the value false.

Example. To store the halting mode for IEEE_OVERFLOW, do a calculation without halting, and restore the halting mode later:
USE, INTRINSIC :: IEEE_ARITHMETIC

LOGICAL HALTING

... 
CALL IEEE_GET_HALTING_MODE (IEEE_OVERFLOW, HALTING) ! Store halting mode
CALL IEEE_SET_HALTING_MODE (IEEE_OVERFLOW, .FALSE.) ! No halting
... ! calculation without halting
CALL IEEE_SET_HALTING_MODE (IEEE_OVERFLOW, HALTING) ! Restore halting mode

17.11.7 IEEE_GET_MODES (MODES)

1 Description. Get floating-point modes.

2 Class. Subroutine.

3 Argument. MODES shall be a scalar of type IEEE_MODES_TYPE. It is an INTENT (OUT) argument that is assigned the value of the floating-point modes.

4 Example. To save the floating-point modes, do a calculation with specific rounding and underflow modes, and restore them later:

USE, INTRINSIC :: IEEE_ARITHMETIC
TYPE (IEEE_MODES_TYPE) SAVE_MODES

... 
CALL IEEE_GET_MODES (SAVE_MODES) ! Save all modes.
CALL IEEE_SET_ROUNDING_MODE (IEEE_TO_ZERO))
CALL IEEE_SET_UNDERFLOW_MODE (GRADUAL=.FALSE.)
... ! calculation with abrupt round-to-zero.
CALL IEEE_SET_MODES (SAVE_MODES) ! Restore all modes.

17.11.8 IEEE_GET_ROUNDING_MODE (ROUND_VALUE [, RADIX])

1 Description. Get rounding mode.

2 Class. Subroutine.

3 Arguments.

ROUND_VALUE shall be a scalar of type IEEE_ROUND_TYPE. It is an INTENT (OUT) argument. It is assigned the value IEEE_NEAREST, IEEE_TO_ZERO, IEEE_UP, IEEE_DOWN, or IEEE_-AWAY if the corresponding rounding mode is in operation and IEEE_OTHER otherwise.

RADIX (optional) shall be an integer scalar with the value two or ten. If RADIX is present with the value ten, the rounding mode queried is the decimal rounding mode, otherwise it is the binary rounding mode.

4 Example. To save the binary rounding mode, do a calculation with round to nearest, and restore the rounding mode later:

USE, INTRINSIC :: IEEE_ARITHMETIC
TYPE (IEEE_ROUND_TYPE) ROUND_VALUE

... 
CALL IEEE_GET_ROUNDING_MODE (ROUND_VALUE) ! Store the rounding mode
CALL IEEE_SET_ROUNDING_MODE (IEEE_NEAREST)
... ! calculation with round to nearest
CALL IEEE_SET_ROUNDING_MODE (ROUND_VALUE) ! Restore the rounding mode

17.11.9 IEEE_GET_STATUS (STATUS_VALUE)

1 Description. Get floating-point status.
2 Class. Subroutine.
3 Argument. STATUS_VALUE shall be a scalar of type IEEE_STATUS_TYPE. It is an INTENT (OUT) argument. It is assigned the value of the floating-point status.
4 Example. To store all the exception flags, do a calculation involving exception handling, and restore them later:

USE, INTRINSIC :: IEEE_ARITHMETIC
TYPE (IEEE_STATUS_TYPE) STATUS_VALUE
... CALL IEEE_GET_STATUS (STATUS_VALUE) ! Get the flags
CALL IEEE_SET_FLAG (IEEE_ALL, .FALSE.) ! Set the flags quiet.
... ! calculation involving exception handling
CALL IEEE_SET_STATUS (STATUS_VALUE) ! Restore the flags

17.11.10 IEEE_GET_UNDERFLOW_MODE (GRADUAL)

1 Description. Get underflow mode.
2 Class. Subroutine.
3 Argument. GRADUAL shall be a logical scalar. It is an INTENT (OUT) argument. It is assigned the value true if the underflow mode is gradual underflow, and false if the underflow mode is abrupt underflow.
4 Restriction. IEEE_GET_UNDERFLOW_MODE shall not be invoked unless IEEE_SUPPORT_UNDERFLOW_CONTROL (X) is true for some X.
5 Example. After CALL IEEE_SET_UNDERFLOW_MODE (.FALSE.), a subsequent CALL IEEE_GET_UNDERFLOW_MODE (GRADUAL) will set GRADUAL to false.

17.11.11 IEEE_INT (A, ROUND [, KIND])

1 Description. Conversion to integer type.
2 Class. Elemental function.
3 Arguments.
A shall be of type real.
ROUND shall be of type IEEE_ROUND_TYPE.
KIND (optional) shall be a scalar integer constant expression.
4 Restriction. IEEE_INT (A, ROUND, KIND) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the value false.
5 Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default integer.
6 Result Value. The result has the value specified by ISO/IEC/IEEE 60559:2011 for the convertToInteger{round} or the convertToIntegerExact{round} operation; the processor shall consistently choose which operation it
provides. That is, the value of A is converted to an integer according to the rounding mode specified by ROUND; if this value is representable in the representation method of the result, the result has this value, otherwise IEEE_INVALID is signaled and the result is processor dependent. If the processor provides the convertToIntegerExact operation, IEEE_INVALID did not signal, and the value of the result differs from that of A, IEEE_INEXACT will be signaled.

Example. The value of IEEE_INT (12.5, IEEE_UP) is 13; IEEE_INEXACT will be signaled if the processor provides the convertToIntegerExact operation.

17.11.12 IEEE_IS_FINITE (X)

Description. Whether a value is finite.

Class. Elemental function.

Argument. X shall be of type real.

Restriction. IEEE_IS_FINITE (X) shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the value false.

Result Characteristics. Default logical.

Result Value. The result has the value true if the value of X is finite, that is, IEEE_CLASS (X) has one of the values IEEE_NEGATIVE_NORMAL, IEEE_NEGATIVE_SUBNORMAL, IEEE_NEGATIVE_ZERO, IEEE_POSITIVE_ZERO, IEEE_POSITIVE_SUBNORMAL, or IEEE_POSITIVE_NORMAL; otherwise, the result has the value false.

Example. IEEE_IS_FINITE (1.0) has the value true.

17.11.13 IEEE_IS_NAN (X)

Description. Whether a value is an IEEE NaN.

Class. Elemental function.

Argument. X shall be of type real.

Restriction. IEEE_IS_NAN (X) shall not be invoked if IEEE_SUPPORT_NAN (X) has the value false.

Result Characteristics. Default logical.

Result Value. The result has the value true if the value of X is an IEEE NaN; otherwise, it has the value false.

Example. IEEE_IS_NAN (SQRT (−1.0)) has the value true if IEEE_SUPPORT_SQRT (1.0) has the value true.

17.11.14 IEEE_IS_NEGATIVE (X)

Description. Whether a value is negative.

Class. Elemental function.

Argument. X shall be of type real.

Restriction. IEEE_IS_NEGATIVE (X) shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the value false.

Result Characteristics. Default logical.

Result Value. The result has the value true if IEEE_CLASS (X) has one of the values IEEE_NEGATIVE_-
NORMAL, IEEE\_NEGATIVE\_SUBNORMAL, IEEE\_NEGATIVE\_ZERO or IEEE\_NEGATIVE\_INF; otherwise, the result has the value false.

Example. IEEE\_IS\_NEGATIVE (0.0) has the value false.

17.11.15 IEEE\_IS\_NORMAL (X)

Description. Whether a value is a normal number.

Class. Elemental function.

Argument. X shall be of type real.

Restriction. IEEE\_IS\_NORMAL (X) shall not be invoked if IEEE\_SUPPORT\_DATATYPE (X) has the value false.

Result Characteristics. Default logical.

Result Value. The result has the value true if IEEE\_CLASS (X) has one of the values IEEE\_NEGATIVE\_NORMAL, IEEE\_NEGATIVE\_ZERO, IEEE\_POSITIVE\_ZERO or IEEE\_POSITIVE\_NORMAL; otherwise, the result has the value false.

Example. IEEE\_IS\_NORMAL (SQRT (−1.0) has the value false if IEEE\_SUPPORT\_SQRT (1.0) has the value true.

17.11.16 IEEE\_LOGB (X)

Description. Exponent.

Class. Elemental function.

Argument. X shall be of type real.

Restriction. IEEE\_LOGB (X) shall not be invoked if IEEE\_SUPPORT\_DATATYPE (X) has the value false.

Result Characteristics. Same as X.

Result Value.

Case (i): If the value of X is neither zero, infinity, nor NaN, the result has the value of the unbiased exponent of X. Note: this value is equal to EXPONENT (X)−1.

Case (ii): If X==0, the result is −infinity if IEEE\_SUPPORT\_INF (X) is true and −HUGE (X) otherwise; IEEE\_DIVIDE\_BY\_ZERO signals.

Case (iii): If IEEE\_SUPPORT\_INF (X) is true and X is infinite, the result is +infinity.

Case (iv): If IEEE\_SUPPORT\_NAN (X) is true and X is a NaN, the result is a NaN.

Example. IEEE\_LOGB (−1.1) has the value 0.0.

17.11.17 IEEE\_MAX\_NUM (X, Y)

Description. Maximum numeric value.

Class. Elemental function.

Arguments.

X shall be of type real.

Y shall be of the same type and kind type parameter as X.

Restriction. IEEE\_MAX\_NUM shall not be invoked if IEEE\_SUPPORT\_DATATYPE (X) has the value false.
Result Characteristics. Same as X.

Result Value. The result has the value specified for the maxNum operation in ISO/IEC/IEEE 60559:2011; that is,

- if \( X < Y \) the result has the value of \( Y \);
- if \( Y < X \) the result has the value of \( X \);
- if exactly one of \( X \) and \( Y \) is a quiet NaN the result has the value of the other argument;
- if both \( X \) and \( Y \) are quiet NaNs the result is either \( X \) or \( Y \) (processor dependent);
- if one or both of \( X \) and \( Y \) are signaling NaNs, IEEE_INVALID signals and the result is a NaN.

Except when \( X \) or \( Y \) is a signaling NaN, no exception is signaled.

Example. The value of IEEE_MAX_NUM (1.5, IEEE_VALUE (IEEE_QUIET_NAN)) is 1.5.

17.11.18 IEEE_MAX_NUM_MAG \((X, Y)\)

Description. Maximum magnitude numeric value.

Class. Elemental function.

Arguments.

\( X \) shall be of type real.

\( Y \) shall be of the same type and kind type parameter as \( X \).

Restriction. IEEE_MAX_NUM_MAG shall not be invoked if IEEE_SUPPORT_DATATYPE \((X)\) has the value false.

Result Characteristics. Same as \( X \).

Result Value. The result has the value specified for the maxNumMag operation in ISO/IEC/IEEE 60559:2011; that is,

- if \(|X| < |Y|\) the result has the value of \( Y \);
- if \(|Y| < |X|\) the result has the value of \( X \);
- if exactly one of \( X \) and \( Y \) is a quiet NaN the result has the value of the other argument;
- if both \( X \) and \( Y \) are quiet NaNs the result is either \( X \) or \( Y \) (processor dependent);
- if one or both of \( X \) and \( Y \) are signaling NaNs, IEEE_INVALID signals and the result is a NaN.

Except when \( X \) or \( Y \) is a signaling NaN, no exception is signaled.

Example. The value of IEEE_MAX_NUM_MAG \((1.5, -2.5)\) is \(-2.5\).

17.11.19 IEEE_MIN_NUM \((X, Y)\)

Description. Minimum numeric value.

Class. Elemental function.

Arguments.

\( X \) shall be of type real.

\( Y \) shall be of the same type and kind type parameter as \( X \).

Restriction. IEEE_MIN_NUM shall not be invoked if IEEE_SUPPORT_DATATYPE \((X)\) has the value false.

Result Characteristics. Same as \( X \).

Result Value.
The result has the value specified for the minNum operation in ISO/IEC/IEEE 60559:2011; that is,

- if \( X < Y \) the result has the value of \( X \);
- if \( Y < X \) the result has the value of \( Y \);
- if exactly one of \( X \) and \( Y \) is a quiet NaN the result has the value of the other argument;
- if both \( X \) and \( Y \) are quiet NaNs the result is either \( X \) or \( Y \) (processor dependent);
- if one or both of \( X \) and \( Y \) are signaling NaNs, IEEE_INVALID signals and the result is a NaN.

Except when \( X \) or \( Y \) is a signaling NaN, no exception is signaled.

Example. The value of IEEE_MIN_NUM (1.5, IEEE_VALUE (IEEE_QUIET_NAN)) is 1.5.

17.11.20 IEEE_MIN_NUM_MAG \((X, Y)\)

Description. Minimum magnitude numeric value.

Class. Elemental function.

Arguments.

\( X \) shall be of type real.
\( Y \) shall be of the same type and kind type parameter as \( X \).

Restriction. IEEE_MIN_NUM_MAG shall not be invoked if IEEE_SUPPORT_DATATYPE (\( X \)) has the value false.

Result Characteristics. Same as \( X \).

Result Value. The result has the value specified for the minNumMag operation in ISO/IEC/IEEE 60559:2011; that is,

- if \( \text{ABS}(X) < \text{ABS}(Y) \) the result has the value of \( X \);
- if \( \text{ABS}(Y) < \text{ABS}(X) \) the result has the value of \( Y \);
- if exactly one of \( X \) and \( Y \) is a quiet NaN the result has the value of the other argument;
- if both \( X \) and \( Y \) are quiet NaNs the result is either \( X \) or \( Y \) (processor dependent);
- if one or both of \( X \) and \( Y \) are signaling NaNs, IEEE_INVALID signals and the result is a NaN.

Except when \( X \) or \( Y \) is a signaling NaN, no exception is signaled.

Example. The value of IEEE_MIN_NUM_MAG (1.5, \(-2.5\)) is 1.5.

17.11.21 IEEE_NEXT_AFTER \((X, Y)\)

Description. Adjacent machine number.

Class. Elemental function.

Arguments. The arguments shall be of type real.

Restriction. IEEE_NEXT_AFTER \((X, Y)\) shall not be invoked if IEEE_SUPPORT_DATATYPE \((X)\) or IEEE_SUPPORT_DATATYPE \((Y)\) has the value false.

Result Characteristics. Same as \( X \).

Result Value.

Case (i): If \( X == Y \), the result is \( X \) and no exception is signaled.

Case (ii): If \( X \neq Y \), the result has the value of the next representable neighbor of \( X \) in the direction of \( Y \). The neighbors of zero (of either sign) are both nonzero. IEEE_OVERFLOW is signaled when
X is finite but IEEE_NEXT_AFTER (X, Y) is infinite; IEEE_UNDERFLOW is signaled when
IEEE_NEXT_AFTER (X, Y) is subnormal; in both cases, IEEE_INEXACT signals.

Example. The value of IEEE_NEXT_AFTER (1.0, 2.0) is 1.0 + EPSILON (X).

17.11.22 IEEE_NEXT_DOWN (X)

Description. Adjacent lower machine number.

Class. Elemental function.

Argument. X shall be of type real.

Restriction. IEEE_NEXT_DOWN (X) shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the
value false. IEEE_NEXT_DOWN (−HUGE (X)) shall not be invoked if IEEE_SUPPORT_INF (X) has the
value false.

Result Characteristics. Same as X.

Result Value. The result has the value specified for the nextDown operation in ISO/IEC/IEEE 60559:2011;
that is, it is the greatest value in the representation method of X that compares less than X, except when X is
equal to −∞ the result has the value −∞, and when X is a NaN the result is a NaN. If X is a signaling NaN,
IEEE_INVALID signals; otherwise, no exception is signaled.

Example. If IEEE_SUPPORT_SUBNORMAL (0.0) is true, the value of IEEE_NEXT_DOWN (+0.0) is the
negative subnormal number with least magnitude.

17.11.23 IEEE_NEXT_UP (X)

Description. Adjacent higher machine number.

Class. Elemental function.

Argument. X shall be of type real.

Restriction. IEEE_NEXT_UP (X) shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the
value false. IEEE_NEXT_UP (HUGE (X)) shall not be invoked if IEEE_SUPPORT_INF (X) has the value false.

Result Characteristics. Same as X.

Result Value. The result has the value specified for the nextUp operation in ISO/IEC/IEEE 60559:2011; that
is, it is the least value in the representation method of X that compares greater than X, except when X is
equal to +∞ the result has the value +∞, and when X is a NaN the result is a NaN. If X is a signaling NaN,
IEEE_INVALID signals; otherwise, no exception is signaled.

Example. If IEEE_SUPPORT_INF (X) is true, the value of IEEE_NEXT_UP (HUGE (X)) is +∞.

17.11.24 IEEE QUIET_EQ (A, B)

Description. Quiet compares equal.

Class. Elemental function.

Arguments.

A shall be of type real.

B shall have the same type and kind type parameter as A.

Restriction. IEEE QUIET_EQ (A, B) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the
value false.
Result Characteristics. Default logical.

Result Value. The result has the value specified for the compareQuietEqual operation in ISO/IEC/IEEE 60559:2011; that is, it is true if and only if A compares equal to B. If A or B is a NaN, the result will be false. If A or B is a signaling NaN, IEEE_INVALID signals; otherwise, no exception is signaled.

Example. IEEE_QUIET_EQ (1.0, IEEE_VALUE (IEEE_QUIET_NAN)) has the value false and no exception is signaled.

17.11.25 IEEE_QUIET_GE (A, B)

Description. Quiet compares greater than or equal.

Class. Elemental function.

Arguments.

A shall be of type real.
B shall have the same type and kind type parameter as A.

Restriction. IEEE_QUIET_GE (A, B) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the value false.

Result Characteristics. Default logical.

Result Value. The result has the value specified for the compareQuietGreaterEqual operation in ISO/IEC/IEEE 60559:2011; that is, it is true if and only if A compares greater than or equal to B. If A or B is a NaN, the result will be false. If A or B is a signaling NaN, IEEE_INVALID signals; otherwise, no exception is signaled.

Example. IEEE_QUIET_GE (1.0, IEEE_VALUE (IEEE_QUIET_NAN)) has the value false and no exception is signaled.

17.11.26 IEEE_QUIET_GT (A, B)

Description. Quiet compares greater than.

Class. Elemental function.

Arguments.

A shall be of type real.
B shall have the same type and kind type parameter as A.

Restriction. IEEE_QUIET_GT (A, B) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the value false.

Result Characteristics. Default logical.

Result Value. The result has the value specified for the compareQuietGreater operation in ISO/IEC/IEEE 60559:2011; that is, it is true if and only if A compares greater than B. If A or B is a NaN, the result will be false. If A or B is a signaling NaN, IEEE_INVALID signals; otherwise, no exception is signaled.

Example. IEEE_QUIET_GT (1.0, IEEE_VALUE (IEEE_QUIET_NAN)) has the value false and no exception is signaled.

17.11.27 IEEE_QUIET_LE (A, B)

Description. Quiet compares less than or equal.

Class. Elemental function.
Arguments.
A shall be of type real.
B shall have the same type and kind type parameter as A.

Restriction. IEEE_QUIET_LE (A, B) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the value false.

Result Characteristics. Default logical.

Result Value. The result has the value specified for the compareQuietLessEqual operation in ISO/IEC/IEEE 60559:2011; that is, it is true if and only if A compares less than or equal to B. If A or B is a NaN, the result will be false. If A or B is a signaling NaN, IEEE_INVALID signals; otherwise, no exception is signaled.

Example. IEEE_QUIET_LE (1.0, IEEE_VALUE (IEEE_QUIET_NAN)) has the value false and no exception is signaled.

17.11.28 IEEE_QUIET_LT (A, B)

Description. Quiet compares less than.

Class. Elemental function.

Arguments.
A shall be of type real.
B shall have the same type and kind type parameter as A.

Restriction. IEEE_QUIET_LT (A, B) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the value false.

Result Characteristics. Default logical.

Result Value. The result has the value specified for the compareQuietLess operation in ISO/IEC/IEEE 60559:2011; that is, it is true if and only if A compares less than B. If A or B is a NaN, the result will be false. If A or B is a signaling NaN, IEEE_INVALID signals; otherwise, no exception is signaled.

Example. IEEE_QUIET_LT (1.0, IEEE_VALUE (IEEE_QUIET_NAN)) has the value false and no exception is signaled.

17.11.29 IEEE_QUIET_NE (A, B)

Description. Quiet compares not equal.

Class. Elemental function.

Arguments.
A shall be of type real.
B shall have the same type and kind type parameter as A.

Restriction. IEEE_QUIET_NE (A, B) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the value false.

Result Characteristics. Default logical.

Result Value. The result has the value specified for the compareQuietNotEqual operation in ISO/IEC/IEEE 60559:2011; that is, it is true if and only if A compares not equal to B. If A or B is a NaN, the result will be true. If A or B is a signaling NaN, IEEE_INVALID signals; otherwise, no exception is signaled.

Example. IEEE_QUIET_NE (1.0, IEEE_VALUE (IEEE_QUIET_NAN)) has the value true and no exception is signaled.
17.11.30  **IEEE_REAL (A [, KIND])**

1. **Description.** Conversion to real type.
2. **Class.** Elemental function.

3. **Arguments.**
   - A shall be of type integer or real.
   - KIND (optional) shall be a scalar integer constant expression.

4. **Restriction.** IEEE_REAL shall not be invoked if A is of type real and IEEE_SUPPORT_DATATYPE (A) has the value false, or if IEEE_SUPPORT_DATATYPE (IEEE_REAL (A, KIND)) has the value false.

5. **Result Characteristics.** Real. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default real.

6. **Result Value.** The result has the same value as A if that value is representable in the representation method of the result, and is rounded according to the rounding mode otherwise. This shall be consistent with the specification of ISO/IEC/IEEE 60559:2011 for the convertFromInt operation when A is of type integer, and with the convertFormat operation otherwise.

7. **Example.** The value of IEEE_REAL (123) is 123.0.

17.11.31  **IEEE_REM (X, Y)**

1. **Description.** Exact remainder.
2. **Class.** Elemental function.

3. **Arguments.** The arguments shall be of type real and have the same radix.

4. **Restriction.** IEEE_REM (X, Y) shall not be invoked if IEEE_SUPPORT_DATATYPE (X) or IEEE_SUPPORT_DATATYPE (Y) has the value false.

5. **Result Characteristics.** Real with the kind type parameter of whichever argument has the greater precision.

6. **Result Value.** The result value, regardless of the rounding mode, shall be exactly \( X - Y \times N \), where \( N \) is the integer nearest to the exact value \( X/Y \); whenever \( |N - X/Y| = \frac{1}{2} \), \( N \) shall be even. If the result value is zero, the sign shall be that of \( X \). This function computes the remainder operation specified in ISO/IEC/IEEE 60559:2011.

7. **Examples.** The value of IEEE_REM (4.0, 3.0) is 1.0, the value of IEEE_REM (3.0, 2.0) is \(-1.0\), and the value of IEEE_REM (5.0, 2.0) is 1.0.

17.11.32  **IEEE_RINT (X [, ROUND])**

1. **Description.** Round to integer.
2. **Class.** Elemental function.

3. **Arguments.**
   - X shall be of type real.
   - ROUND (optional) shall be of type IEEE_ROUND_TYPE.

4. **Restriction.** IEEE_RINT (X) shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the value false.

5. **Result Characteristics.** Same as X.
Result Value. If ROUND is present, the value of the result is the value of X rounded to an integer according to the mode specified by ROUND; this is the ISO/IEC/IEEE 60559:2011 operation roundToInteger{rounding}. Otherwise, the value of the result is that specified for the operation roundIntegralToExact in ISO/IEC/IEEE 60559:2011; this is the value of X rounded to an integer according to the rounding mode. If the result has the value zero, the sign is that of X.

Examples. If the rounding mode is round to nearest, the value of IEEE_RINT (1.1) is 1.0. The value of IEEE_RINT (1.1, IEEE_UP) is 2.0.

17.11.33 IEEE_SCALB (X, I)

Description. $X \times 2^I$.

Class. Elemental function.

Arguments.

- $X$ shall be of type real.
- $I$ shall be of type integer.

Restriction. IEEE_SCALB (X) shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the value false.

Result Characteristics. Same as $X$.

Result Value.

Case (i): If $X \times 2^I$ is representable as a normal number, the result has this value.

Case (ii): If $X$ is finite and $X \times 2^I$ is too large, the IEEE_OVERFLOW exception shall occur. If IEEE_-SUPPORT_INF (X) is true, the result value is infinity with the sign of X; otherwise, the result value is SIGN (HUGE (X), X).

Case (iii): If $X \times 2^I$ is too small and there is loss of accuracy, the IEEE_UNDERFLOW exception shall occur. The result is the representable number having a magnitude nearest to $|2^I|$ and the same sign as $X$.

Case (iv): If $X$ is infinite, the result is the same as $X$; no exception signals.

Example. The value of IEEE_SCALB (1.0, 2) is 4.0.

17.11.34 IEEE_SELECTED_REAL_KIND ([P, R, RADIX])

Description. IEEE kind type parameter value.

Class. Transformational function.

Arguments. At least one argument shall be present.

- $P$ (optional) shall be an integer scalar.
- $R$ (optional) shall be an integer scalar.
- RADIX (optional) shall be an integer scalar.

Result Characteristics. Default integer scalar.

Result Value. If $P$ or $R$ is absent, the result value is the same as if it were present with the value zero. If RADIX is absent, there is no requirement on the radix of the selected kind. The result has a value equal to a value of the kind type parameter of an ISO/IEC/IEEE 60559:2011 floating-point format with decimal precision, as returned by the intrinsic function PRECISION, of at least $P$ digits, a decimal exponent range, as returned by the intrinsic function RANGE, of at least $R$, and a radix, as returned by the intrinsic function RADIX, of RADIX, if such a kind type parameter is available on the processor.

Otherwise, the result is $-1$ if the processor supports an IEEE real type with radix RADIX and exponent range of at least $R$ but not with precision of at least $P$, $-2$ if the processor supports an IEEE real type with radix
RADIX and precision of at least P but not with exponent range of at least R, −3 if the processor supports an
IEEE real type with radix RADIX but with neither precision of at least P nor exponent range of at least R, −4 if
the processor supports an IEEE real type with radix RADIX and either precision of at least P or exponent range
of at least R but not both together, and −5 if the processor supports no IEEE real type with radix RADIX.

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.

Example. IEEE_SELECTED_REAL_KIND (6, 30) has the value KIND (0.0) on a machine that supports

17.11.35 IEEE_SET_FLAG (FLAG, FLAG_VALUE)

Description. Set an exception flag.

Class. Pure subroutine.

Arguments.

FLAG shall be a scalar or array of type IEEE_FLAG_TYPE. If a value of FLAG is IEEE_INVALID,
IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_UNDERFLOW, or IEEE_INEXACT,
the corresponding exception flag is assigned a value. No two elements of FLAG shall have the same
value.

FLAG_VALUE shall be a logical scalar or array. It shall be conformable with FLAG. If an element has the value
true, the corresponding flag is set to be signaling; otherwise, the flag is set to be quiet.

Example. CALL IEEE_SET_FLAG (IEEE_OVERFLOW, .TRUE.) sets the IEEE_OVERFLOW flag to be
signaling.

17.11.36 IEEE_SET_HALTING_MODE (FLAG, HALTING)

Description. Set a halting mode.

Class. Pure subroutine.

Arguments.

FLAG shall be a scalar or array of type IEEE_FLAG_TYPE. It shall have only the values IEEE_-
INVALID, IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_UNDERFLOW, or IEEE_--
INEXACT. No two elements of FLAG shall have the same value.

HALTING shall be a logical scalar or array. It shall be conformable with FLAG. If an element has the value
ture, the corresponding exception specified by FLAG will cause halting. Otherwise, execution will
continue after this exception.

Restriction. IEEE_SET_HALTING_MODE (FLAG, HALTING) shall not be invoked if IEEE_SUPPORT_--
HALTING (FLAG) has the value false.

Example. CALL IEEE_SET_HALTING_MODE (IEEE_DIVIDE_BY_ZERO, .TRUE.) causes halting after
a divide_by_zero exception.

17.11.37 IEEE_SET_MODES (MODES)

Description. Set floating-point modes.

Class. Subroutine.

Argument. MODES shall be a scalar of type IEEE_MODES_TYPE. Its value shall be one that was assigned
by a previous invocation of IEEE_GET_MODES to its MODES argument. The floating-point modes (17.7) are
restored to the state at that invocation.
Example. To save the floating-point modes, do a calculation with specific rounding and underflow modes, and restore them later:

```
USE, INTRINSIC :: IEEE_ARITHMETIC

TYPE (IEEE_MODES_TYPE) SAVE_MODES

... 
CALL IEEE_GET_MODES (SAVE_MODES) ! Save all modes.
CALL IEEE_SET_ROUNDING_MODE (IEEE_TO_ZERO))
CALL IEEE_SET_UNDERFLOW_MODE (GRADUAL=.FALSE.)
... ! calculation with abrupt round-to-zero.
CALL IEEE_SET_MODES (SAVE_MODES) ! Restore all modes.
```

17.11.38 IEEE_SET_ROUNDING_MODE (ROUND_VALUE [, RADIX])

**Description.** Set rounding mode.

**Class.** Subroutine.

**Arguments.**

- ROUND_VALUE shall be a scalar of type IEEE_ROUND_TYPE. It specifies the rounding mode to be set.
- RADIX (optional) shall be an integer scalar with the value two or ten. If RADIX is present with the value ten, the rounding mode set is the decimal rounding mode; otherwise it is the binary rounding mode.

**Restriction.** IEEE_SET_ROUNDING_MODE (ROUND_VALUE) shall not be invoked unless IEEE_SUPPORT_ROUNDING (ROUND_VALUE, X) is true for some X such that IEEE_SUPPORT_DATATYPE (X) is true. IEEE_SET_ROUNDING_MODE (ROUND_VALUE, RADIX) shall not be invoked unless IEEE_SUPPORT_ROUNDING (ROUND_VALUE, X) is true for some X with radix RADIX such that IEEE_SUPPORT_DATATYPE (X) is true.

**Example.** To save the binary rounding mode, do a calculation with round to nearest, and restore the rounding mode later:

```
USE, INTRINSIC :: IEEE_ARITHMETIC

TYPE (IEEE_ROUND_TYPE) ROUND_VALUE

... 
CALL IEEE_GET_ROUNDING_MODE (ROUND_VALUE) ! Store the rounding mode
CALL IEEE_SET_ROUNDING_MODE (IEEE_NEAREST)
... ! calculation with round to nearest
CALL IEEE_SET_ROUNDING_MODE (ROUND_VALUE) ! Restore the rounding mode
```

17.11.39 IEEE_SET_STATUS (STATUS_VALUE)

**Description.** Restore floating-point status.

**Class.** Subroutine.

**Argument.** STATUS_VALUE shall be a scalar of type IEEE_STATUS_TYPE. Its value shall be one that was assigned by a previous invocation of IEEE_GET_STATUS to its STATUS_VALUE argument. The floating-point status (17.7 is restored to the state at that invocation).

**Example.** To store all the exceptions flags, do a calculation involving exception handling, and restore them later:

```
USE, INTRINSIC :: IEEE_ARITHMETIC

TYPE (IEEE_ROUND_TYPE) ROUND_VALUE

... 
CALL IEEE_GET_ROUNDING_MODE (ROUND_VALUE) ! Store the rounding mode
CALL IEEE_SET_ROUNDING_MODE (IEEE_NEAREST)
... ! calculation with round to nearest
CALL IEEE_SET_ROUNDING_MODE (ROUND_VALUE) ! Restore the rounding mode
```

470 ISO/IEC JTC 1/SC 22/WG5/N2118
USE, INTRINSIC :: IEEE_EXCEPTIONS

TYPE (IEEE_STATUS_TYPE) STATUS_VALUE

CALL IEEE_GET_STATUS (STATUS_VALUE) ! Store the flags
CALL IEEE_SET_FLAG (IEEE_ALL, .FALSE.) ! Set them quiet
! calculation involving exception handling
CALL IEEE_SET_STATUS (STATUS_VALUE) ! Restore the flags

17.11.40 IEEE_SET_UNDERFLOW_MODE (GRADUAL)

Description. Set underflow mode.

Class. Subroutine.

Argument. GRADUAL shall be a logical scalar. If it is true, the underflow mode is set to gradual underflow. If it is false, the underflow mode is set to abrupt underflow.

Restriction. IEEE_SET_UNDERFLOW_MODE shall not be invoked unless IEEE_SUPPORT_UNDERFLOW_CONTROL (X) is true for some X.

Example. To perform some calculations with abrupt underflow and then restore the previous mode:

USE, INTRINSIC :: IEEE_ARITHMETIC
LOGICAL SAVE_UNDERFLOW_MODE

CALL IEEE_GET_UNDERFLOW_MODE (SAVE_UNDERFLOW_MODE)
CALL IEEE_SET_UNDERFLOW_MODE (GRADUAL=.FALSE.)
! Perform some calculations with abrupt underflow
CALL IEEE_SET_UNDERFLOW_MODE (SAVE_UNDERFLOW_MODE)

17.11.41 IEEE_SIGNALING_EQ (A, B)

Description. Signaling compares equal.

Class. Elemental function.

Arguments. A shall be of type real. B shall be of the same type and kind type parameter as A.

Restriction. IEEE_SIGNALING_EQ (A, B) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the value false.

Result Characteristics. Default logical.

Result Value. The result has the value specified for the compareSignalingEqual operation in ISO/IEC/IEEE 60559:2011; that is, it is true if and only if A compares equal to B. If A or B is a NaN, the result will be false and IEEE_INVALID signals; otherwise, no exception is signaled.

Example. IEEE_SIGNALING_EQ (1.0, IEEE_VALUE (IEEE_QUIET_NAN)) has the value false and signals IEEE_INVALID.
17.11.42 IEEE_SIGNALING_GE (A, B)

Description. Signaling compares greater than or equal.

Class. Elemental function.

Arguments.

A shall be of type real.
B shall be of the same type and kind type parameter as A.

Restriction. IEEE_SIGNALING_GE (A, B) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the value false.

Result Characteristics. Default logical.

Result Value. The result has the value specified for the compareSignalingGreaterEqual operation in ISO/IEC/IEEE 60559:2011; that is, it is true if and only if A compares greater than or equal to B. If A or B is a NaN, the result will be false and IEEE_INVALID signals; otherwise, no exception is signaled.

Example. IEEE_SIGNALING_GE (1.0, IEEE_VALUE (IEEE_QUIET_NAN)) has the value false and signals IEEE_INVALID.

17.11.43 IEEE_SIGNALING_GT (A, B)

Description. Signaling compares greater than.

Class. Elemental function.

Arguments.

A shall be of type real.
B shall be of the same type and kind type parameter as A.

Restriction. IEEE_SIGNALING_GT (A, B) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the value false.

Result Characteristics. Default logical.

Result Value. The result has the value specified for the compareSignalingGreater operation in ISO/IEC/IEEE 60559:2011; that is, it is true if and only if A compares greater than B. If A or B is a NaN, the result will be false and IEEE_INVALID signals; otherwise, no exception is signaled.

Example. IEEE_SIGNALING_GT (1.0, IEEE_VALUE (IEEE_QUIET_NAN)) has the value false and signals IEEE_INVALID.

17.11.44 IEEE_SIGNALING_LE (A, B)

Description. Signaling compares less than or equal.

Class. Elemental function.

Arguments.

A shall be of type real.
B shall be of the same type and kind type parameter as A.

Restriction. IEEE_SIGNALING_LE (A, B) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the value false.

Result Characteristics. Default logical.
Result Value. The result has the value specified for the compareSignalingLessEqual operation in ISO/IEC/IEEE 60559:2011; that is, it is true if and only if A compares less than or equal to B. If A or B is a NaN, the result will be false and IEEE_INVALID signals; otherwise, no exception is signaled.

Example. IEEE_SIGNALING_LE (1.0, IEEE_VALUE (IEEE_QUIET_NAN)) has the value false and signals IEEE_INVALID.

17.11.45 IEEE_SIGNALING_LT (A, B)

Description. Signaling compares less than.

Class. Elemental function.

Arguments.

A shall be of type real.

B shall be of the same type and kind type parameter as A.

Restriction. IEEE_SIGNALING_LT (A, B) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the value false.

Result Characteristics. Default logical.

Result Value. The result has the value specified for the compareSignalingLess operation in ISO/IEC/IEEE 60559:2011; that is, it is true if and only if A compares less than B. If A or B is a NaN, the result will be false and IEEE_INVALID signals; otherwise, no exception is signaled.

Example. IEEE_SIGNALING_LT (1.0, IEEE_VALUE (IEEE_QUIET_NAN)) has the value false and signals IEEE_INVALID.

17.11.46 IEEE_SIGNALING_NE (A, B)

Description. Signaling compares not equal.

Class. Elemental function.

Arguments.

A shall be of type real.

B shall be of the same type and kind type parameter as A.

Restriction. IEEE_SIGNALING_NE (A, B) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the value false.

Result Characteristics. Default logical.

Result Value. The result has the value specified for the compareSignalingNotEqual operation in ISO/IEC/IEEE 60559:2011; that is, it is true if and only if A compares not equal to B. If A or B is a NaN, the result will be true and IEEE_INVALID signals; otherwise, no exception is signaled.

Example. IEEE_SIGNALING_NE (1.0, IEEE_VALUE (IEEE_QUIET_NAN)) has the value true and signals IEEE_INVALID.

17.11.47 IEEE_SIGNBIT (X)

Description. Test sign bit.

Class. Elemental function.

Argument. X shall be of type real.
4 **Restriction.** IEEE_SIGNBIT (X) shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the value false.

5 **Result Characteristics.** Default logical.

6 **Result Value.** The result has the value specified for the isSignMinus operation in ISO/IEC/IEEE 60559:2011; that is, it is true if and only if the sign bit of X is nonzero. No exception is signaled even if X is a signaling NaN.

7 **Example.** IEEE_SIGNBIT (−1.0) has the value true.

17.11.48 **IEEE_SUPPORT_DATATYPE () or IEEE_SUPPORT_DATATYPE (X)**

1 **Description.** Query IEEE arithmetic support.

2 **Class.** Inquiry function.

3 **Argument.** X shall be of type real. It may be a scalar or an array.

4 **Result Characteristics.** Default logical scalar.

5 **Result Value.** The result has the value true if the processor supports IEEE arithmetic for all reals (X does not appear) or for real variables of the same kind type parameter as X; otherwise, it has the value false. Here, support is as defined in the first paragraph of 17.9.

6 **Example.** If default real kind conforms to ISO/IEC/IEEE 60559:2011 except that underflow values flush to zero instead of being subnormal, IEEE_SUPPORT_DATATYPE (1.0) has the value true.

17.11.49 **IEEE_SUPPORT_DENORMAL () or IEEE_SUPPORT_DENORMAL (X)**

1 **Description.** Query subnormal number support.

2 **Class.** Inquiry function.

3 **Argument.** X shall be of type real. It may be a scalar or an array.

4 **Result Characteristics.** Default logical scalar.

5 **Result Value.**

Case (i): IEEE_SUPPORT_DENORMAL (X) has the value true if IEEE_SUPPORT_DATATYPE (X) has the value true and the processor supports arithmetic operations and assignments with subnormal numbers (biased exponent e = 0 and fraction f ≠ 0, see subclause 3.2 of ISO/IEC/IEEE 60559:2011) for real variables of the same kind type parameter as X; otherwise, it has the value false.

Case (ii): IEEE_SUPPORT_DENORMAL () has the value true if IEEE_SUPPORT_DENORMAL (X) has the value true for all real X; otherwise, it has the value false.

6 **Example.** IEEE_SUPPORT_DENORMAL (X) has the value true if the processor supports subnormal values for X.

**NOTE 17.10**
A reference to IEEE_SUPPORT_DENORMAL will have the same result value as a reference to IEEE_SUPPORT_SUBNORMAL with the same argument list.
17.11.50 IEEE_SUPPORT_DIVIDE () or IEEE_SUPPORT_DIVIDE (X)

1 Description. Query IEEE division support.
2 Class. Inquiry function.
3 Argument. X shall be of type real. It may be a scalar or an array.
4 Result Characteristics. Default logical scalar.

5 Result Value.

Case (i): IEEE_SUPPORT_DIVIDE (X) has the value true if the processor supports division with the
accuracy specified by ISO/IEC/IEEE 60559:2011 for real variables of the same kind type parameter
as X; otherwise, it has the value false.

Case (ii): IEEE_SUPPORT_DIVIDE () has the value true if IEEE_SUPPORT_DIVIDE (X) has the value
true for all real X; otherwise, it has the value false.

6 Example. IEEE_SUPPORT_DIVIDE (X) has the value true if division of operands with the same kind as X

17.11.51 IEEE_SUPPORT_FLAG (FLAG) or IEEE_SUPPORT_FLAG (FLAG, X)

1 Description. Query exception support.
2 Class. Transformational function.
3 Arguments.

FLAG shall be a scalar of type IEEE_FLAG_TYPE. Its value shall be one of IEEE_INVALID, IEEE_-
OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_UNDERFLOW, or IEEE_INEXACT.

X shall be of type real. It may be a scalar or an array.

4 Result Characteristics. Default logical scalar.

5 Result Value.

Case (i): IEEE_SUPPORT_FLAG (FLAG, X) has the value true if the processor supports detection of the
specified exception for real variables of the same kind type parameter as X; otherwise, it has the
value false.

Case (ii): IEEE_SUPPORT_FLAG (FLAG) has the value true if IEEE_SUPPORT_FLAG (FLAG, X) has the value
true for all real X; otherwise, it has the value false.

6 Example. IEEE_SUPPORT_FLAG (IEEE_INEXACT) has the value true if the processor supports the inexact
exception.

17.11.52 IEEE_SUPPORT_HALTING (FLAG)

1 Description. Query halting mode support.
2 Class. Transformational function.
3 Argument. FLAG shall be a scalar of type IEEE_FLAG_TYPE. Its value shall be one of IEEE_INVALID,
IEEE OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_UNDERFLOW, or IEEE_INEXACT.
4 Result Characteristics. Default logical scalar.

5 Result Value. The result has the value true if the processor supports the ability to control during program
execution whether to abort or continue execution after the exception specified by FLAG; otherwise, it has the
value false. Support includes the ability to change the mode by CALL IEEE_SET_HALTING_MODE (FLAG).
Example. IEEE_SUPPORT_HALTING (IEEE_OVERFLOW) has the value true if the processor supports control of halting after an overflow.

17.11.53 IEEE_SUPPORT_INF () or IEEE_SUPPORT_INF (X)

1 Description. Query IEEE infinity support.
2 Class. Inquiry function.
3 Argument. X shall be of type real. It may be a scalar or an array.
4 Result Characteristics. Default logical scalar.
5 Result Value.

Case (i): IEEE_SUPPORT_INF (X) has the value true if the processor supports IEEE infinities (positive and negative) for real variables of the same kind type parameter as X; otherwise, it has the value false.

Case (ii): IEEE_SUPPORT_INF () has the value true if IEEE_SUPPORT_INF (X) has the value true for all real X; otherwise, it has the value false.

Example. IEEE_SUPPORT_INF (X) has the value true if the processor supports IEEE infinities for X.

17.11.54 IEEE_SUPPORT_IO () or IEEE_SUPPORT_IO (X)

1 Description. Query IEEE formatting support.
2 Class. Inquiry function.
3 Argument. X shall be of type real. It may be a scalar or an array.
4 Result Characteristics. Default logical scalar.
5 Result Value.

Case (i): IEEE_SUPPORT_IO (X) has the value true if base conversion during formatted input/output (12.5.6.16, 12.6.2.13, 13.7.2.3.8) conforms to ISO/IEC/IEEE 60559:2011 for the modes UP, DOWN, ZERO, and NEAREST for real variables of the same kind type parameter as X; otherwise, it has the value false.

Case (ii): IEEE_SUPPORT_IO () has the value true if IEEE_SUPPORT_IO (X) has the value true for all real X; otherwise, it has the value false.

Example. IEEE_SUPPORT_IO (X) has the value true if formatted input/output base conversions conform to ISO/IEC/IEEE 60559:2011.

17.11.55 IEEE_SUPPORT_NAN () or IEEE_SUPPORT_NAN (X)

1 Description. Query IEEE NaN support.
2 Class. Inquiry function.
3 Argument. X shall be of type real. It may be a scalar or an array.
4 Result Characteristics. Default logical scalar.
5 Result Value.

Case (i): IEEE_SUPPORT_NAN (X) has the value true if the processor supports IEEE NaNs for real variables of the same kind type parameter as X; otherwise, it has the value false.

Case (ii): IEEE_SUPPORT_NAN () has the value true if IEEE_SUPPORT_NAN (X) has the value true for all real X; otherwise, it has the value false.
Example. IEEE_SUPPORT_NAN (X) has the value true if the processor supports IEEE NaNs for X.

**17.11.56** IEEE_SUPPORT_ROUNDING (ROUND_VALUE) or 
IEEE_SUPPORT_ROUNDING (ROUND_VALUE, X)

1 Description. Query IEEE rounding support.

2 Class. Transformational function.

3 Arguments.

   ROUND_VALUE shall be of type IEEE_ROUND_TYPE.

   X shall be of type real. It may be a scalar or an array.

4 Result Characteristics. Default logical scalar.

5 Result Value.

   Case (i): IEEE_SUPPORT_ROUNDING (ROUND_VALUE, X) has the value true if the processor supports 
   the rounding mode defined by ROUND_VALUE for real variables of the same kind type parameter 
   as X; otherwise, it has the value false. Support includes the ability to change the mode by CALL 
   IEEE_SET_ROUNDING_MODE (ROUND_VALUE).

   Case (ii): IEEE_SUPPORT_ROUNDING (ROUND_VALUE) has the value true if IEEE_SUPPORT_- 
   ROUNDING (ROUND_VALUE, X) has the value true for all real X; otherwise, it has the value 
   false.

Example. IEEE_SUPPORT_ROUNDING (IEEE_TO_ZERO) has the value true if the processor supports 
rounding to zero for all reals.

**17.11.57** IEEE_SUPPORT_SQRT () or IEEE_SUPPORT_SQRT (X)

1 Description. Query IEEE square root support.

2 Class. Inquiry function.

3 Argument. X shall be of type real. It may be a scalar or an array.

4 Result Characteristics. Default logical scalar.

5 Result Value.

   Case (i): IEEE_SUPPORT_SQRT (X) has the value true if the intrinsic function SQRT conforms to 
   ISO/IEC/IEEE 60559:2011 for real variables of the same kind type parameter as X; otherwise, 
   it has the value false.

   Case (ii): IEEE_SUPPORT_SQRT () has the value true if IEEE_SUPPORT_SQRT (X) has the value true 
   for all real X; otherwise, it has the value false.

Example. If IEEE_SUPPORT_SQRT (1.0) has the value true, SQRT (−0.0) will have the value −0.0.

**17.11.58** IEEE_SUPPORT_STANDARD () or 
IEEE_SUPPORT_STANDARD (X)

1 Description. Query IEEE standard support.

2 Class. Inquiry function.

3 Argument. X shall be of type real. It may be a scalar or an array.

4 Result Characteristics. Default logical scalar.
Result Value.

Case (i): IEEE_SUPPORT_STANDARD (X) has the value true if the results of all the functions IEEE_SUPPORT_DATATYPE (X), IEEE_SUPPORT_DIVIDE (X), IEEE_SUPPORT_FLAG (FLAG, X) for valid FLAG, IEEE_SUPPORT_HALTING (FLAG) for valid FLAG, IEEE_SUPPORT_INF (X), IEEE_SUPPORT_NAN (X), IEEE_SUPPORT_ROUNDING (ROUND_VALUE, X) for valid ROUND_VALUE, IEEE_SUPPORT_SQRT (X), and IEEE_SUPPORT_SUBNORMAL (X) are all true; otherwise, it has the value false.

Case (ii): IEEE_SUPPORT_standard () has the value true if IEEE_SUPPORT_STANDARD (X) has the value true for all real X; otherwise, it has the value false.

Example. IEEE_SUPPORT_STANDARD () has the value false if some but not all kinds of reals conform to ISO/IEC/IEEE 60559:2011.

17.11.59 IEEE_SUPPORT_SUBNORMAL () or IEEE_SUPPORT_SUBNORMAL (X)

Description. Query subnormal number support.

Class. Inquiry function.

Argument. X shall be of type real. It may be a scalar or an array.

Result Characteristics. Default logical scalar.

Result Value.

Case (i): IEEE_SUPPORT_SUBNORMAL (X) has the value true if IEEE_SUPPORT_DATATYPE (X) has the value true and the processor supports arithmetic operations and assignments with subnormal numbers (biased exponent $e = 0$ and fraction $f \neq 0$, see subclause 3.2 of ISO/IEC/IEEE 60559:2011) for real variables of the same kind type parameter as X; otherwise, it has the value false.

Case (ii): IEEE_SUPPORT_SUBNORMAL () has the value true if IEEE_SUPPORT_SUBNORMAL (X) has the value true for all real X; otherwise, it has the value false.

Example. IEEE_SUPPORT_SUBNORMAL (X) has the value true if the processor supports subnormal values for X.

NOTE 17.11

The subnormal numbers are not included in the 16.4 model for real numbers; they satisfy the inequality \( \text{ABS} (X) < \text{TINY} (X) \). They usually occur as a result of an arithmetic operation whose exact result is less than \( \text{TINY} (X) \). Such an operation causes IEEE_UNDERFLOW to signal unless the result is exact. IEEE_SUPPORT_SUBNORMAL (X) is false if the processor never returns a subnormal number as the result of an arithmetic operation.

17.11.60 IEEE_SUPPORT_UNDERFLOW_CONTROL () or IEEE_SUPPORT_UNDERFLOW_CONTROL (X)

Description. Query underflow control support.

Class. Inquiry function.

Argument. X shall be of type real. It may be a scalar or an array.

Result Characteristics. Default logical scalar.

Result Value.

Case (i): IEEE_SUPPORT_UNDERFLOW_CONTROL (X) has the value true if the processor supports control of the underflow mode for floating-point calculations with the same type as X, and false otherwise.
Case (ii): IEEE_SUPPORT_UNDERFLOW_CONTROL () has the value true if the processor supports control of the underflow mode for all floating-point calculations, and false otherwise.

Example. IEEE_SUPPORT_UNDERFLOW_CONTROL (2.5) has the value true if the processor supports underflow mode control for default real calculations.

17.11.61 IEEE_UNORDERED (X, Y)

Description. Whether two values are unordered.

Class. Elemental function.

Arguments. The arguments shall be of type real.

Restriction. IEEE_UNORDERED (X, Y) shall not be invoked if IEEE_SUPPORT_DATATYPE (X) or IEEE_SUPPORT_DATATYPE (Y) has the value false.

Result Characteristics. Default logical.

Result Value. The result has the value true if X or Y is a NaN or both are NaNs; otherwise, it has the value false. If X or Y is a signaling NaN, IEEE_INVALID may signal.

Example. IEEE_UNORDERED (0.0, SQRT (−1.0)) has the value true if IEEE_SUPPORT_SQRT (1.0) has the value true.

17.11.62 IEEE_VALUE (X, CLASS)

Description. Return number in a class.

Class. Elemental function.

Arguments.

X shall be of type real.

CLASS shall be of type IEEE_CLASS_TYPE. The value is permitted to be: IEEE_SIGNALING_NAN or IEEE_QUIET_NAN if IEEE_SUPPORT_NAN (X) has the value true, IEEE_NEGATIVE_INF or IEEE_POSITIVE_INF if IEEE_SUPPORT_INF (X) has the value true, IEEE_NEGATIVE_SUBNORMAL or IEEE_POSITIVE_SUBNORMAL if IEEE_SUPPORT_SUBNORMAL (X) has the value true, IEEE_NEGATIVE_NORMAL, IEEE_NEGATIVE_ZERO, IEEE_POSITIVE_ZERO or IEEE_POSITIVE_NORMAL.

Restriction. IEEE_VALUE (X, CLASS) shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the value false.

Result Characteristics. Same as X.

Result Value. The result value is an IEEE value as specified by CLASS. Although in most cases the value is processor dependent, the value shall not vary between invocations for any particular X kind type parameter and CLASS value.

Example. IEEE_VALUE (1.0, IEEE_NEGATIVE_INF) has the value −infinity.

Whenever IEEE_VALUE returns a signaling NaN, it is processor dependent whether or not invalid is raised and processor dependent whether or not the signaling NaN is converted into a quiet NaN.

NOTE 17.12

If the expr in an assignment statement is a reference to the IEEE_VALUE function that returns a signaling NaN and the variable is of the same type and kind as the function result, it is recommended that the signaling NaN be preserved.
17.12 Examples

NOTE 17.13

MODULE DOT
   ! Module for dot product of two real arrays of rank 1.
   ! The caller needs to ensure that exceptions do not cause halting.
   USE, INTRINSIC :: IEEE_EXCEPTIONS
   LOGICAL :: MATRIX_ERROR = .FALSE.
   INTERFACE OPERATOR(.dot.)
      MODULE PROCEDURE MULT
   END INTERFACE
   CONTAINS
      REAL FUNCTION MULT (A, B)
      REAL, INTENT (IN) :: A(:), B(:)
      INTEGER I
      LOGICAL OVERFLOW
      IF (SIZE(A) /= SIZE(B)) THEN
         MATRIX_ERROR = .TRUE.
         RETURN
      END IF
      ! The processor ensures that IEEE_OVERFLOW is quiet.
      MULT = 0.0
      DO I = 1, SIZE (A)
         MULT = MULT + A(I)*B(I)
      END DO
      CALL IEEE_GET_FLAG (IEEE_OVERFLOW, OVERFLOW)
      IF (OVERFLOW) MATRIX_ERROR = .TRUE.
   END FUNCTION MULT
END MODULE DOT

This module provides a function that computes the dot product of two real arrays of rank 1. If the sizes of the arrays are different, an immediate return occurs with MATRIX_ERROR true. If overflow occurs during the actual calculation, the IEEE_OVERFLOW flag will signal and MATRIX_ERROR will be true.

NOTE 17.14

USE, INTRINSIC :: IEEE_EXCEPTIONS
USE, INTRINSIC :: IEEE_FEATURES, ONLY: IEEE_INVALID_FLAG
! The other exceptions of IEEE_USUAL (IEEE_OVERFLOW and
! IEEE_DIVIDE_BY_ZERO) are always available with IEEE_EXCEPTIONS
TYPE (IEEE_STATUS_TYPE) STATUS_VALUE
   LOGICAL, DIMENSION(3) :: FLAG_VALUE
   ...
   CALL IEEE_GET_FLAG (IEEE_OVERFLOW, OVERFLOW)
   IF (OVERFLOW) MATRIX_ERROR = .TRUE.
END MODULE DOT

CALL IEEE_SET_HALTING_MODE (IEEE_USUAL, .FALSE.) ! Needed in case the
! default on the processor is to halt on exceptions
CALL IEEE_SET_FLAG (IEEE_USUAL, .FALSE.)
NOTE 17.14 (cont.)

! First try the "fast" algorithm for inverting a matrix:
MATRIX1 = FAST_INV (MATRIX) ! This shall not alter MATRIX.
CALL IEEE_GET_FLAG (IEEE_USUAL, FLAG_VALUE)
IF (ANY(FLAG_VALUE)) THEN
  ! "Fast" algorithm failed; try "slow" one:
  CALL IEEE_SET_FLAG (IEEE_USUAL, .FALSE.)
  MATRIX1 = SLOW_INV (MATRIX)
  CALL IEEE_GET_FLAG (IEEE_USUAL, FLAG_VALUE)
  IF (ANY (FLAG_VALUE)) THEN
    WRITE (*, *) 'Cannot invert matrix'
    STOP
  END IF
END IF
CALL IEEE_SET_STATUS (STATUS_VALUE)

In this example, the function FAST_INV might cause a condition to signal. If it does, another try is made with SLOW_INV. If this still fails, a message is printed and the program stops. Note, also, that it is important to set the flags quiet before the second try. The state of all the flags is stored and restored.

NOTE 17.15

USE, INTRINSIC :: IEEE_EXCEPTIONS
LOGICAL FLAG_VALUE
...
CALL IEEE_SET_HALTING_MODE (IEEE_OVERFLOW, .FALSE.)
! First try a fast algorithm for inverting a matrix.
CALL IEEE_SET_FLAG (IEEE_OVERFLOW, .FALSE.)
DO K = 1, N
  ...
  CALL IEEE_GET_FLAG (IEEE_OVERFLOW, FLAG_VALUE)
  IF (FLAG_VALUE) EXIT
END DO
IF (FLAG_VALUE) THEN
  ! Alternative code which knows that K-1 steps have executed normally.
  ...
END IF

Here the code for matrix inversion is in line and the transfer is made more precise by adding extra tests of the flag.
18 Interoperability with C

18.1 General

Fortran provides a means of referencing procedures that are defined by means of the C programming language or procedures that can be described by C prototypes as defined in 6.7.6.3 of ISO/IEC 9899:2011, even if they are not actually defined by means of C. Conversely, there is a means of specifying that a procedure defined by a Fortran subprogram can be referenced from a function defined by means of C. In addition, there is a means of declaring global variables that are associated with C variables whose names have external linkage as defined in 6.2.2 of ISO/IEC 9899:2011.

The ISO_C_BINDING module provides access to named constants that represent kind type parameters of data representations compatible with C types. Fortran also provides facilities for defining derived types (7.5) and enumerations (7.6) that correspond to C types.

The source file ISO_Fortran_binding.h provides definitions and prototypes to enable a C function to interoperate with a Fortran procedure that has a dummy data object that is allocatable, assumed-shape, assumed-rank, pointer, or is of type character with an assumed length.

18.2 The ISO_C_BINDING intrinsic module

18.2.1 Summary of contents

The processor shall provide the intrinsic module ISO_C_BINDING. This module shall make accessible the following entities: the named constants C_NULL_PTR, C_NULL_FUNPTR, and those with names listed in the first column of Table 15.1 and the second column of Table 15.2, the types C_PTR and C_FUNPTR, and the procedures in 18.2.3. A processor may provide other public entities in the ISO_C_BINDING intrinsic module in addition to those listed here.

18.2.2 Named constants and derived types in the module

The entities listed in the second column of Table 18.2 shall be default integer named constants.

A Fortran intrinsic type whose kind type parameter is one of the values in the module shall have the same representation as the C type with which it interoperates, for each value that a variable of that type can have. For C_BOOL, the internal representation of \_TRUE\_C_BOOL and \_FALSE\_C_BOOL shall be the same as those of the C values \_Bool\_1 and \_Bool\_0 respectively.

The value of C\_INT shall be a valid value for an integer kind type parameter on the processor. The values of C\_SHORT, C\_LONG, C\_LONG\_LONG, C\_SIGNED\_CHAR, C\_SIZE\_T, C\_INT8\_T, C\_INT16\_T, C\_INT32\_T, C\_INT64\_T, C\_INT\_LEAST8\_T, C\_INT\_LEAST16\_T, C\_INT\_LEAST32\_T, C\_INT\_LEAST64\_T, C\_INT\_FAST8\_T, C\_INT\_FAST16\_T, C\_INT\_FAST32\_T, C\_INT\_FAST64\_T, C\_INTMAX\_T, and C\_INTPTR\_T shall each be a valid value for an integer kind type parameter on the processor or shall be −1 if the companion processor (5.5.7) does not define the corresponding C type and there is no interoprating Fortran processor kind, or −2 if the companion processor does not define the corresponding C type.

The values of C\_FLOAT, C\_DOUBLE, and C\_LONG\_DOUBLE shall each be a valid value for a real kind type parameter on the processor or shall be −1 if the companion processor’s type does not have a precision equal to the precision of any of the Fortran processor’s real kinds, −2 if the companion processor’s type does not have a range equal to the range of any of the Fortran processor’s real kinds, −3 if the companion processor’s type
has neither the precision nor range of any of the Fortran processor’s real kinds, and equal to −4 if there is no
interoperating Fortran processor kind for other reasons. The values of C_FLOAT_COMPLEX, C_DOUBLE-
COMPLEX, and C_LONG_DOUBLE_COMPLEX shall be the same as those of C_FLOAT, C_DOUBLE, and
C_LONG_DOUBLE, respectively.

The value of C_BOOL shall be a valid value for a logical kind parameter on the processor or shall be −1.

The value of C_CHAR shall be a valid value for a character kind type parameter on the processor or shall be −1.
If the value of C_CHAR is nonnegative, the character kind specified is the C character kind; otherwise, there is
no C character kind.

The following entities shall be named constants of type character with a length parameter of one. The kind
parameter value shall be equal to the value of C_CHAR unless C_CHAR = −1, in which case the kind parameter
value shall be the same as for default kind. The values of these constants are specified in Table 18.1. In the case
that C_CHAR ≠ −1 the value is specified using C syntax. The semantics of these values are explained in 5.2.1
and 5.2.2 of ISO/IEC 9899:2011.

Table 18.1: Names of C characters with special semantics

<table>
<thead>
<tr>
<th>Name</th>
<th>C definition</th>
<th>C_CHAR = −1</th>
<th>C_CHAR ≠ −1</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_NULL_CHAR</td>
<td>null character</td>
<td>CHAR(0)</td>
<td>‘\0’</td>
</tr>
<tr>
<td>C_ALERT</td>
<td>alert</td>
<td>ACHAR(7)</td>
<td>‘\a’</td>
</tr>
<tr>
<td>C_BACKSPACE</td>
<td>backspace</td>
<td>ACHAR(8)</td>
<td>‘\b’</td>
</tr>
<tr>
<td>C_FORM_FEED</td>
<td>form feed</td>
<td>ACHAR(12)</td>
<td>‘\f’</td>
</tr>
<tr>
<td>C_NEW_LINE</td>
<td>new line</td>
<td>ACHAR(10)</td>
<td>‘\n’</td>
</tr>
<tr>
<td>C_CARRIAGE_RETURN</td>
<td>carriage return</td>
<td>ACHAR(13)</td>
<td>‘\r’</td>
</tr>
<tr>
<td>C_HORIZONTAL_TAB</td>
<td>horizontal tab</td>
<td>ACHAR(9)</td>
<td>‘\t’</td>
</tr>
<tr>
<td>C_VERTICAL_TAB</td>
<td>vertical tab</td>
<td>ACHAR(11)</td>
<td>‘\v’</td>
</tr>
</tbody>
</table>

The entities C_PTR and C_FUNPTR are described in 18.3.3.

The entity C_NULL_PTR shall be a named constant of type C_PTR. The value of C_NULL_PTR shall be the
same as the value NULL in C. The entity C_NULL_FUNPTR shall be a named constant of type C_FUNPTR.
The value of C_NULL_FUNPTR shall be that of a null pointer to a function in C.

NOTE 18.1
The value of NEW_LINE(C_NEW_LINE) is C_NEW_LINE (16.9.140).

18.2.3 Procedures in the module

18.2.3.1 General

In the detailed descriptions below, procedure names are generic and not specific. The C_F_POINTER subroutine
is impure; all other procedures in the module are pure.

18.2.3.2 C_ASSOCIATED (C_PTR_1 [, C_PTR_2])

Description. Query C pointer status.

Class. Transformational function.

Arguments.
C_PTR_1 shall be a scalar of type C_PTR or C_FUNPTR.
C_PTR_2 (optional) shall be a scalar of the same type as C_PTR_1.
4 Result Characteristics. Default logical scalar.

5 Result Value.

Case (i): If C_PTR_2 is absent, the result is false if C_PTR_1 is a C null pointer and true otherwise.

Case (ii): If C_PTR_2 is present, the result is false if C_PTR_1 is a C null pointer. If C_PTR_1 is not a C null pointer, the result is true if C_PTR_1 compares equal to C_PTR_2 in the sense of 6.3.2.3 and 6.5.9 of ISO/IEC 9899:2011, and false otherwise.

NOTE 18.2
The following example illustrates the use of C_LOC and C_ASSOCIATED.

USE, INTRINSIC :: ISO_C_BINDING, ONLY: C_PTR, C_FLOAT, C_ASSOCIATED, C_LOC
INTERFACE
  SUBROUTINE FOO(GAMMA) BIND(C)
    IMPORT C_PTR
    TYPE(C_PTR), VALUE :: GAMMA
  END SUBROUTINE FOO
END INTERFACE
REAL(C_FLOAT), TARGET, DIMENSION(100) :: ALPHA
TYPE(C_PTR) :: BETA
...
IF (.NOT. C_ASSOCIATED(BETA)) THEN
  BETA = C_LOC(ALPHA)
ENDIF
CALL FOO(BETA)

18.2.3.3 C_F_POINTER (CPTR, FPTR [, SHAPE])

Description. Associate a data pointer with the target of a C pointer and specify its shape.

Class. Subroutine.

Arguments.

CPTR shall be a scalar of type C_PTR. It is an INTENT (IN) argument. Its value shall be

- the C address of an interoperable data entity,
- the result of a reference to C_LOC with a noninteroperable argument, or
- the C address of a storage sequence that is not in use by any other Fortran entity.

The value of CPTR shall not be the C address of a Fortran variable that does not have the TARGET attribute.

FPTR shall be a pointer, shall not have a deferred type parameter, and shall not be a coindexed object. It is an INTENT (OUT) argument. If FPTR is an array, its shape is specified by SHAPE and each lower bound is equal to 1.

Case (i): If the value of CPTR is the C address of an interoperable data entity, FPTR shall be a data pointer with type and type parameter values interoperable with the type of the entity. If the target T of CPTR is scalar, FPTR becomes pointer associated with T; if FPTR is an array, SHAPE shall specify a size of 1. If T is an array, and FPTR is scalar, FPTR becomes associated with the first element of T. If both T and FPTR are arrays, SHAPE shall specify a size that is less than or equal to the size of T, and FPTR becomes associated with the first PRODUCT (SHAPE) elements of T (this could be the entirety of T).
Case (ii): If the value of CPTR is the result of a reference to C_LOC with a noninter-operable effective argument X, FPTR shall be a nonpolymorphic pointer with the same type and type parameters as X. In this case, X shall not have been deallocated or have become undefined due to execution of a RETURN or END statement since the reference. If X is scalar, FPTR becomes pointer associated with X; if FPTR is an array, SHAPE shall specify a size of 1. If X is an array and FPTR is scalar, FPTR becomes associated with the first element of X. If both X and FPTR are arrays, SHAPE shall specify a size that is less than or equal to the size of X, and FPTR becomes associated with the first PRODUCT (SHAPE) elements of X (this could be the entirety of X).

Case (iii): If the value of CPTR is the C address of a storage sequence that is not in use by any other Fortran entity, FPTR becomes associated with that storage sequence. If FPTR is an array, its shape is specified by SHAPE and each lower bound is 1. The storage sequence shall be large enough to contain the target object described by FPTR and shall satisfy any other processor-dependent requirement for association.

SHAPE (optional) shall be a rank-one integer array. It is an INTENT (IN) argument. SHAPE shall be present if and only if FPTR is an array; its size shall be equal to the rank of FPTR.

4 Examples.

Case (i): extern double c_x;
void *address_of_x (void)
{
    return &c_x;
}

! Assume interface to "address_of_x" is available.
Real (C_double), Pointer :: xp
Call C_F_Pointer (address_of_x (), xp)

Case (ii): Type t
    Real, Allocatable :: v(:,:)
End Type
Type(t), Target :: x
Type(C_ptr) :: xloc
xloc = C_Loc (x)
...
Type(t), Pointer :: y
Call C_F_Pointer (xloc, y)

Case (iii): void *getmem (int nbits)
{
    return malloc ((nbits+CHAR_BIT-1)/CHAR_BIT);
}

! Assume interface to "getmem" is available,
! and there is a derived type "mytype" accessible.
Type(mytype), Pointer :: x
Call C_F_Pointer (getmem (Storage_Size (x)), x)

The following statements illustrate the use of C_F_POINTER when the pointer to be set has a deferred type.
 parameter:
     Character(42), Pointer :: C1
     Character(:), Pointer :: C2
 Call C_F_Pointer (CPTR, C1)
     C2 => C1

This will associate C2 with the entity at the C address specified by CPTR, and specify its length to be the same as that of C1.

NOTE 18.3
In the case of associating FPTR with a storage sequence, there might be processor-dependent requirements such as alignment of the memory address or placement in memory.

18.2.3.4 C_F_PROCPOINTER (CPTR, FPTR)

1 Description. Associate a procedure pointer with the target of a C function pointer.

2 Class. Pure subroutine.

3 Arguments.
   CPTR shall be a scalar of type C_FUNPTR. It is an INTENT (IN) argument. Its value shall be the C address of a procedure that is interoperable, or the result of a reference to the function C_FUNLOC from the intrinsic module ISO_C_BINDING.

   FPTR shall be a procedure pointer, and shall not be a component of a coindexed object. It is an INTENT (OUT) argument. If the target of CPTR is interoperable, the interface for FPTR shall be interoperable with the prototype that describes the target of CPTR; otherwise, the interface for FPTR shall have the same characteristics as that target. FPTR becomes pointer associated with the target of CPTR.

NOTE 18.4
The term “target” in the descriptions of C_F_POINTER and C_F_PROCPOINTER denotes the entity referenced by a C pointer, as described in 6.2.5 of ISO/IEC 9899:2011.

18.2.3.5 C_FUNLOC (X)

1 Description. C address of the argument.

2 Class. Transformational function.

3 Argument. X shall be a procedure; if it is a procedure pointer it shall be associated. It shall not be a coindexed object.

4 Result Characteristics. Scalar of type C_FUNPTR.

5 Result Value. The result value is described using the result name FUNPTR. The result is determined as if C_FUNPTR were a derived type containing a procedure pointer component PX with an implicit interface and the pointer assignment FUNPTR%PX => X were executed.

6 The result is a value that can be used as an actual CPTR argument in a call to C_F_PROCPOINTER where the FPTR argument has attributes that would allow the pointer assignment FPTR => X. Such a call to C_F_PROCPOINTER shall have the effect of the pointer assignment FPTR => X.
18.2.3.6 C_LOC (X)

Description. C address of the argument.

Class. Transformational function.

Argument. X shall have either the POINTER or TARGET attribute. It shall not be a coindexed object. It shall either be a variable with interoperable type and kind type parameters, or be a nonpolymorphic variable with no length type parameters. If it is allocatable, it shall be allocated. If it is a pointer, it shall be associated. If it is an array, it shall be contiguous and have nonzero size. It shall not be a zero-length string.

Result Characteristics. Scalar of type C_PTR.

Result Value. The result value is described using the result name CPTR.

If X is a scalar data entity, the result is determined as if C_PTR were a derived type containing a scalar pointer component PX of the type and type parameters of X and the pointer assignment CPTR%PX => X were executed.

If X is an array data entity, the result is determined as if C_PTR were a derived type containing a scalar pointer component PX of the type and type parameters of X and the pointer assignment of CPTR%PX to the first element of X were executed.

If X is a data entity that is interoperable or has interoperable type and type parameters, the result is the value that the C processor returns as the result of applying the unary “&” operator (as defined in ISO/IEC 9899:2011, 6.5.3.2) to the target of CPTR%PX.

The result is a value that can be used as an actual CPTR argument in a call to C_F_POINTER where FPTR has attributes that would allow the pointer assignment FPTR => X. Such a call to C_F_POINTER shall have the effect of the pointer assignment FPTR => X.

NOTE 18.5
Where the actual argument is of noninteroperable type or type parameters, the result of C_LOC provides an opaque “handle” for it. In an actual implementation, this handle might be the C address of the argument; however, only a C function that treats it as a void (generic) C pointer that cannot be dereferenced (6.5.3.2 in ISO/IEC 9899:2011) is likely to be portable.

18.2.3.7 C_SIZEOF (X)

Description. Size of X in bytes.

Class. Inquiry function.

Argument. X shall be an interoperable data entity that is not an assumed-size array.

Result Characteristics. Scalar integer of kind C_SIZE_T (18.3.2).

Result Value. If X is scalar, the result value is the value that the companion processor returns as the result of applying the sizeof operator (ISO/IEC 9899:2011, subclause 6.5.3.4) to an object of a type that interoperates with the type and type parameters of X.

If X is an array, the result value is the value that the companion processor returns as the result of applying the sizeof operator to an object of a type that interoperates with the type and type parameters of X, multiplied by the number of elements in X.
18.3 Interoperability between Fortran and C entities

18.3.1 General

Subclause 18.3 defines the conditions under which a Fortran entity is interoperable. If a Fortran entity is interoperable, an equivalent entity could be defined by means of C and the Fortran entity would interoperate with the C entity. There does not have to be such an interoperating C entity.

NOTE 18.6
A Fortran entity can be interoperable with more than one C entity.

18.3.2 Interoperability of intrinsic types

Table 18.2 shows the interoperability between Fortran intrinsic types and C types. A Fortran intrinsic type with particular type parameter values is interoperable with a C type if the type and kind type parameter value are listed in the table on the same row as that C type. If the type is character, the length type parameter is interoperable if and only if its 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.

The second column of the table refers to the named constants made accessible by the ISO_C_BINDING intrinsic module. If the value of any of these named constants is negative, there is no combination of Fortran type and type parameters interoperable with the C type shown in that row.

A combination of intrinsic type and type parameters is interoperable if it is interoperable with a C type. The C types mentioned in table 18.2 are defined in subclauses 6.2.5, 7.19, and 7.20.1 of ISO/IEC 9899:2011.

Table 18.2: Interoperability between Fortran and C types

<table>
<thead>
<tr>
<th>Fortran type</th>
<th>Named constant from the ISO_C_BINDING module (kind type parameter if value is positive)</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>C_INT</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>C_SHORT</td>
<td>short int</td>
</tr>
<tr>
<td></td>
<td>C_LONG</td>
<td>long int</td>
</tr>
<tr>
<td></td>
<td>C_LONGoplast</td>
<td>long long int</td>
</tr>
<tr>
<td></td>
<td>C_SIGNED_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned char</td>
</tr>
<tr>
<td></td>
<td>C_SIZE_T</td>
<td>size_t</td>
</tr>
<tr>
<td></td>
<td>C_INT8_T</td>
<td>int8_t</td>
</tr>
<tr>
<td></td>
<td>C_INT16_T</td>
<td>int16_t</td>
</tr>
<tr>
<td></td>
<td>C_INT32_T</td>
<td>int32_t</td>
</tr>
<tr>
<td></td>
<td>C_INT64_T</td>
<td>int64_t</td>
</tr>
<tr>
<td></td>
<td>C_INT_LEAST8_T</td>
<td>int_least8_t</td>
</tr>
<tr>
<td></td>
<td>C_INT_LEAST16_T</td>
<td>int_least16_t</td>
</tr>
<tr>
<td></td>
<td>C_INT_LEAST32_T</td>
<td>int_least32_t</td>
</tr>
<tr>
<td></td>
<td>C_INT_LEAST64_T</td>
<td>int_least64_t</td>
</tr>
<tr>
<td></td>
<td>C_INT_FAST8_T</td>
<td>int_fast8_t</td>
</tr>
<tr>
<td></td>
<td>C_INT_FAST16_T</td>
<td>int_fast16_t</td>
</tr>
<tr>
<td></td>
<td>C_INT_FAST32_T</td>
<td>int_fast32_t</td>
</tr>
<tr>
<td></td>
<td>C_INT_FAST64_T</td>
<td>int_fast64_t</td>
</tr>
<tr>
<td></td>
<td>C_INTMAX_T</td>
<td>intmax_t</td>
</tr>
</tbody>
</table>
Interoperability between Fortran and C types  

<table>
<thead>
<tr>
<th>Fortran type</th>
<th>Named constant from the ISO_C_BINDING module (kind type parameter if value is positive)</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL</td>
<td>C_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td></td>
<td>C_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td></td>
<td>C_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>COMPLEX</td>
<td>C_FLOAT_COMPLEX</td>
<td>float _Complex</td>
</tr>
<tr>
<td></td>
<td>C_DOUBLE_COMPLEX</td>
<td>double _Complex</td>
</tr>
<tr>
<td></td>
<td>C_LONG_DOUBLE_COMPLEX</td>
<td>long double _Complex</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>C_BOOL</td>
<td>_Bool</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>C_CHAR</td>
<td>char</td>
</tr>
</tbody>
</table>

NOTE 18.7

ISO/IEC 9899:2011 specifies that the representations for nonnegative signed integers are the same as the corresponding values of unsigned integers. Because Fortran does not provide direct support for unsigned kinds of integers, the ISO_C_BINDING module does not make accessible named constants for their kind type parameter values. A user can use the signed kinds of integers to interoperate with the unsigned types and all their qualified versions as well. This has the potentially surprising side effect that the C type unsigned char is interoperable with the type integer with a kind type parameter of C_SIGNED_CHAR.

18.3.3 Interoperability with C pointer types

C_PTR and C_FUNPTR shall be derived types with only private components. No direct component of either of these types is allocatable or a pointer. C_PTR is interoperable with any C object pointer type. C_FUNPTR is interoperable with any C function pointer type.

NOTE 18.8

This means that only a C processor with the same representation method for all C object pointer types, and the same representation method for all C function pointer types, can be the target of interoperability of a Fortran processor. ISO/IEC 9899:2011 does not require this to be the case.

NOTE 18.9

The function C_LOC can be used to return a value of type C_PTR that is the C address of an allocated allocatable variable. The function C_FUNLOC can be used to return a value of type C_FUNPTR that is the C address of a procedure. For C_LOC and C_FUNLOC the returned value is of an interoperable type and thus can be used in contexts where the procedure or allocatable variable is not directly allowed. For example, it could be passed as an actual argument to a C function.

Similarly, type C_FUNPTR or C_PTR can be used in a dummy argument or structure component and can have a value that is the C address of a procedure or allocatable variable, even in contexts where a procedure or allocatable variable is not directly allowed.

18.3.4 Interoperability of derived types and C struct types

Interoperability between a derived type in Fortran and a struct type in C is provided by the BIND attribute on the Fortran type.

C1801 (R726) A derived type with the BIND attribute shall not have the SEQUENCE attribute.
C1802  (R726) A derived type with the BIND attribute shall not have type parameters.

C1803  (R726) A derived type with the BIND attribute shall not have the EXTENDS attribute.

C1804  (R726) A derived-type-def that defines a derived type with the BIND attribute shall not have a type-bound-procedure-part.

C1805  (R726) A derived type with the BIND attribute shall have at least one component.

C1806  (R726) Each component of a derived type with the BIND attribute shall be a nonpointer, nonallocatable data component with interoperable type and type parameters.

NOTE 18.10
The syntax rules and their constraints require that a derived type that is interoperable with a C struct type have components that are all data entities that are interoperable. No component is permitted to be allocatable or a pointer, but the value of a component of type C_FUNPTR or C_PTR can be the C address of such an entity.

NOTE 18.11
The names of the corresponding components of the derived type and the C struct type need not be the same.

NOTE 18.12
For example, the C type myctype, declared below, is interoperable with the Fortran type myftype, declared below.

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

The names of the types and the names of the components are not significant for the purposes of determining whether a Fortran derived type is interoperable with a C struct type.

NOTE 18.13
ISO/IEC 9899:2011 requires the names and component names to be the same in order for the types to be compatible (ISO/IEC 9899:2011, subclause 6.2.7). This is similar to Fortran's rule describing when different derived type definitions describe the same sequence type. This rule was not extended to determine whether
18.3.5 Interoperability of scalar variables

1 A named scalar Fortran variable is interoperable if and only if its type and type parameters are interoperable, it is not a coarray, it has neither the ALLOCATABLE nor the POINTER attribute, and if it is of type character its length is not assumed or declared by an expression that is not a constant expression.

2 An interoperable scalar Fortran variable is interoperable with a scalar C entity if their types and type parameters are interoperable.

18.3.6 Interoperability of array variables

1 A Fortran variable that is a named array is interoperable if and only if its type and type parameters are interoperable, it is not a coarray, it is of explicit shape or assumed size, and if it is of type character its length is not assumed or declared by an expression that is not a constant expression.

2 An explicit-shape or assumed-size array of rank $r$, with a shape of $[e_1 \ldots e_r]$ is interoperable with a C array if its size is nonzero and

(1) either

(a) the array is assumed-size, and the C array does not specify a size, or

(b) the array is an explicit-shape array, and the extent of the last dimension ($e_r$) is the same as the size of the C array, and

(2) either

(a) $r$ is equal to one, and an element of the array is interoperable with an element of the C array, or

(b) $r$ is greater than one, and an explicit-shape array with shape of $[e_1 \ldots e_{r-1}]$, with the same type and type parameters as the original array, is interoperable with a C array of a type equal to the element type of the original C array.

NOTE 18.13 (cont.)
a Fortran derived type is interoperable with a C struct type because the case of identifiers is significant in C but not in Fortran.

NOTE 18.14

An element of a multi-dimensional C array is an array type, so a Fortran array of rank one is not interoperable with a multidimensional C array.

NOTE 18.15

An allocatable array or array pointer is never interoperable. Such an array does not meet the requirement of being an explicit-shape or assumed-size array.

NOTE 18.16

For example, a Fortran array declared as

```
INTEGER(C_INT) :: A(18, 3:7, *)
```

is interoperable with a C array declared as

```
int b[][5][18];
```
NOTE 18.17
The C programming language defines null-terminated strings, which are actually arrays of the C type char that have a C null character in them to indicate the last valid element. A Fortran array of type character with a kind type parameter equal to C_CHAR is interoperable with a C string.

Fortran’s rules of sequence association (15.5.2.11) permit a character scalar actual argument to correspond to a dummy argument array. This makes it possible to argument associate a Fortran character string with a C string.

Note 18.21 has an example of interoperation between Fortran and C strings.

18.3.7 Interoperability of procedures and procedure interfaces

1 A Fortran procedure is interoperable if it has the BIND attribute, that is, if its interface is specified with a proc-language-binding-spec.

2 A Fortran procedure interface is interoperable with a C function prototype if

   (1) the interface has the BIND attribute,
   (2) either
      (a) the interface describes a function whose result is a scalar variable that is interoperable with the result of the prototype or
      (b) the interface describes a subroutine and the prototype has a result type of void,
   (3) the number of dummy arguments of the interface is equal to the number of formal parameters of the prototype,
   (4) any scalar dummy argument with the VALUE attribute is interoperable with the corresponding formal parameter of the prototype,
   (5) any dummy argument without the VALUE attribute corresponds to a formal parameter of the prototype that is of a pointer type, and either
      • the dummy argument is interoperable with an entity of the referenced type (ISO/IEC 9899:2011, 6.2.5, 7.19, and 7.20.1) of the formal parameter,
      • the dummy argument is a nonallocatable nonpointer variable of type CHARACTER with assumed character length and the formal parameter is a pointer to CFI_cdesc_t,
      • the dummy argument is allocatable, assumed-shape, assumed-rank, or a pointer without the CONTIGUOUS attribute, and the formal parameter is a pointer to CFI_cdesc_t, or
      • the dummy argument is assumed-type and not allocatable, assumed-shape, assumed-rank, or a pointer, and the formal parameter is a pointer to void,
   (6) each allocatable or pointer dummy argument of type CHARACTER has deferred character length, and
   (7) the prototype does not have variable arguments as denoted by the ellipsis (...).

NOTE 18.18
The referenced type of a C pointer type is the C type of the object that the C pointer type points to. For example, the referenced type of the pointer type int * is int.

NOTE 18.19
The C language allows specification of a C function that can take a variable number of arguments (ISO/IEC 9899:2011, 7.16). This document does not provide a mechanism for Fortran procedures to interoperate with such C functions.

A formal parameter of a C function prototype corresponds to a dummy argument of a Fortran interface if they are in the same relative positions in the C parameter list and the dummy argument list, respectively.
In a reference from C to a Fortran procedure with an interoperable interface, a C actual argument shall be the
address of a C descriptor for the intended effective argument if the corresponding dummy argument interoperates
with a C formal parameter that is a pointer to CFI_cdesc_t. In this C descriptor, the members other than
attribute and type shall describe an object with the same characteristics as the intended effective argument.
The value of the attribute member of the C descriptor shall be compatible with the characteristics of the dummy
argument. The type member shall have a value that depends on the intended effective argument as follows:

1. if the dynamic type of the intended effective argument is an interoperable type listed in Table 18.4, the
   corresponding value for that type;
2. if the dynamic type of the intended effective argument is an intrinsic type for which the processor defines
   a nonnegative type specifier value not listed in Table 18.4, that type specifier value;
3. otherwise, CFI_type_other.

When an interoperable Fortran procedure with a simply contiguous dummy argument is invoked from C and the
actual argument is the address of a C descriptor for a discontiguous object, the Fortran processor shall handle
the difference in contiguity.

When an interoperable C procedure whose Fortran interface has a simply contiguous dummy argument is invoked
from Fortran and the effective argument is discontiguous, the Fortran processor shall ensure that the C procedure
receives a descriptor for a contiguous object.

When an interoperable C procedure whose Fortran interface has a simply contiguous dummy argument is invoked
from C, and the actual argument is the address of a C descriptor for a discontiguous object, the C code within
the procedure shall be prepared to handle the discontiguous argument.

If an interoperable procedure defined by means other than Fortran has an optional dummy argument, and the
corresponding actual argument in a reference from Fortran is absent, the procedure is invoked with a null pointer
for that argument. If an interoperable procedure defined by means of Fortran is invoked by a C function, an
optional dummy argument is absent if and only if the corresponding argument in the invocation is a null pointer.

NOTE 18.20

For example, a Fortran procedure interface described by

```fortran
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
```
is interoperable with the C function prototype

```c
short func(int i, double *j, int *k, int l[10], void *m);
```

A C pointer can correspond to a Fortran dummy argument of type C_PTR with the VALUE attribute or
to a Fortran scalar that does not have the VALUE attribute. In the above example, the C pointers j and
k correspond to the Fortran scalars J and K, respectively, and the C pointer m corresponds to the Fortran
dummy argument M of type C_PTR.
NOTE 18.21

The interoperability of Fortran procedure interfaces with C function prototypes is only one part of invocation of a C function from Fortran. There are four pieces to consider in such an invocation: the procedure reference, the Fortran procedure interface, the C function prototype, and the C function. Conversely, the invocation of a Fortran procedure from C involves the function reference, the C function prototype, the Fortran procedure interface, and the Fortran procedure. In order to determine whether a reference is allowed, it is necessary to consider all four pieces.

For example, consider a C function that can be described by the C function prototype

```c
void copy(char in[], char out[]);
```

Such a function can be invoked from Fortran as follows:

```fortran
USE, INTRINSIC :: ISO_C_BINDING, ONLY: C_CHAR, C_NULL_CHAR
INTERFACE
SUBROUTINE COPY(IN, OUT) BIND(C)
  IMPORT C_CHAR
  CHARACTER(KIND=C_CHAR), DIMENSION(*) :: IN, OUT
END SUBROUTINE COPY
END INTERFACE

CHARACTER(LEN=10, KIND=C_CHAR) :: &
  & DIGIT_STRING = C_CHAR_'123456789' // C_NULL_CHAR
CHARACTER(KIND=C_CHAR) :: DIGIT_ARR(10)
CALL COPY(DIGIT_STRING, DIGIT_ARR)
PRINT '(1X, A1)', DIGIT_ARR(1:9)
END
```

The procedure reference has character string actual arguments. These correspond to character array dummy arguments in the procedure interface body as allowed by Fortran’s rules of sequence association (15.5.2.11). Those array dummy arguments in the procedure interface are interoperable with the formal parameters of the C function prototype. The C function is not shown here, but is assumed to be compatible with the C function prototype.

NOTE 18.22

The requirements on the Fortran processor when a discontiguous actual argument is being passed to a simply contiguous dummy argument are such that the C programmer has the same functionality (and safety) as the Fortran programmer, when either the calling or called procedure are Fortran procedures. The requirements on the C procedure, in the case when both the called and calling procedure are C procedures, effectively require that the C procedure not access memory through the descriptor that is not described by the descriptor; otherwise the program will not conform to this document. A dummy argument is simply contiguous if it has the CONTIGUOUS attribute or is an array that is not assumed-shape, assumed-rank, or a pointer.

18.4 C descriptors

A C descriptor is a C structure of type CFI_cdesc_t. Together with library functions that have standard prototypes, it provides a means for describing and manipulating Fortran data objects from within a C function. This C structure is defined in the source file ISO_Fortran_binding.h.
18.5 The source file ISO_Fortran_binding.h

18.5.1 Summary of contents

The source file ISO_Fortran_binding.h shall contain the C structure definitions, typedef declarations, macro definitions, and function prototypes specified in subclauses 18.5.2 to 18.5.5. The definitions and declarations in ISO_Fortran_binding.h can be used by a C function to interpret and manipulate a C descriptor. These provide a means to specify a C prototype that interoperates with a Fortran interface that has a non-interoperable dummy variable (18.3.7).

The source file ISO_Fortran_binding.h may be included in any order relative to the standard C headers, and may be included more than once in a given scope, with no effect different from being included only once, other than the effect on line numbers.

A C source file that includes the ISO_Fortran_binding.h header file shall not use any names starting with CFI_ that are not defined in the header, and shall not define any of the structure names defined in the header as macro names. All names other than structure member names defined in the header begin with CFI_ or an underscore character, or are defined by a standard C header that it includes.

18.5.2 The CFI_dim_t structure type

CFI_dim_t is a typedef name for a C structure. It is used to represent lower bound, extent, and memory stride information for one dimension of an array. The type CFI_index_t is described in 18.5.4. CFI_dim_t contains at least the following members in any order.

- CFI_index_t lower_bound; The value is equal to the value of the lower bound for the dimension being described.
- CFI_index_t extent; The value is equal to the number of elements in the dimension being described, or −1 for the final dimension of an assumed-size array.
- CFI_index_t sm; The value is equal to the memory stride for a dimension; this is the difference in bytes between the addresses of successive elements in the dimension being described.

18.5.3 The CFI_cdesc_t structure type

CFI_cdesc_t is a typedef name for a C structure, which contains a flexible array member. It shall contain at least the members described in this subclause. The values of these members of a structure of type CFI_cdesc_t that is produced by the functions and macros specified in this document, or received by a C function when invoked by a Fortran procedure, shall have the properties described in this subclause.

The first three members of the structure shall be base_addr, elem_len, and version in that order. The final member shall be dim. All other members shall be between version and dim, in any order. The types CFI_ attribute_t, CFI_rank_t, and CFI_type_t are described in 18.5.4. The type CFI_dim_t is described in 18.5.2.

- void * base_addr; If the object is an unallocated allocatable variable or a pointer that is disassociated, the value is a null pointer. If the object has zero size, the value is not a null pointer but is otherwise processor-dependent. Otherwise, the value is the base address of the object being described. The base address of a scalar is its C address. The base address of an array is the C address of the first element in Fortran array element order.

- size_t elem_len; If the object is scalar, the value is the storage size in bytes of the object; otherwise, the value is the storage size in bytes of an element of the object.

- int version; The value is equal to the value of CFI_VERSION in the source file ISO_Fortran_binding.h that defined the format and meaning of this C descriptor when the descriptor was established.
CFI_rank_t rank; The value is equal to the number of dimensions of the Fortran object being described; if the object is scalar, the value is zero.

CFI_type_t type; The value is equal to the specifier for the type of the object. Each interoperable intrinsic C type has a specifier. Specifiers are also provided to indicate that the type of the object is an interoperable structure, or is unknown. The macros listed in Table 18.4 provide values that correspond to each specifier.

CFI_attribute_t attribute; The value is equal to the value of an attribute code that indicates whether the object described is allocatable, a data pointer, or a nonallocatable nonpointer data object. The macros listed in Table 18.3 provide values that correspond to each code.

CFI_dim_t dim; The number of elements in the dim array is equal to the rank of the object. Each element of the array contains the lower bound, extent, and memory stride information for the corresponding dimension of the Fortran object.

For a C descriptor of an array pointer or allocatable array, the value of the lower_bound member of each element of the dim member of the descriptor is determined by argument association, allocation, or pointer association. For a C descriptor of a nonallocatable nonpointer object, the value of the lower_bound member of each element of the dim member of the descriptor is zero.

There shall be an ordering of the dimensions such that the absolute value of the sm member of the first dimension is not less than the elem_len member of the C descriptor and the absolute value of the sm member of each subsequent dimension is not less than the absolute value of the sm member of the previous dimension multiplied by the extent of the previous dimension.

In a C descriptor of an assumed-size array, the extent member of the last element of the dim member has the value −1.

NOTE 18.23
The reason for the restriction on the absolute values of the sm members is to ensure that there is no overlap between the elements of the array that is being described, while allowing for the reordering of subscripts. Within Fortran, such a reordering can be achieved with the intrinsic function TRANSPOSE or the intrinsic function RESHAPE with the optional argument ORDER, and an optimizing compiler can accommodate it without making a copy by constructing the appropriate descriptor whenever it can determine that a copy is not needed.

NOTE 18.24
The value of elem_len for a Fortran CHARACTER object is equal to the character length times the number of bytes of a single character of that kind. If the kind is C_CHAR, this value will be equal to the character length.

18.5.4 Macros and typedefs in ISO_Fortran_binding.h

1 Except for CFI_CDESC_T, each macro defined in ISO_Fortran_binding.h expands to an integer constant expression that is either a single token or a parenthesized expression that is suitable for use in #if preprocessing directives.

2 CFI_CDESC_T is a function-like macro that takes one argument, which is the rank of the C descriptor to create, and evaluates to an unqualified type of suitable size and alignment for defining a variable to use as a C descriptor of that rank. The argument shall be an integer constant expression with a value that is greater than or equal to zero and less than or equal to CFI_MAX_RANK. A pointer to a variable declared using CFI_CDESC_T can be cast to CFI_cdesc_t *. A variable declared using CFI_CDESC_T shall not have an initializer.

NOTE 18.23
The reason for the restriction on the absolute values of the sm members is to ensure that there is no overlap between the elements of the array that is being described, while allowing for the reordering of subscripts. Within Fortran, such a reordering can be achieved with the intrinsic function TRANSPOSE or the intrinsic function RESHAPE with the optional argument ORDER, and an optimizing compiler can accommodate it without making a copy by constructing the appropriate descriptor whenever it can determine that a copy is not needed.

NOTE 18.24
The value of elem_len for a Fortran CHARACTER object is equal to the character length times the number of bytes of a single character of that kind. If the kind is C_CHAR, this value will be equal to the character length.
NOTE 18.25

The CFI_CDESC_T macro provides the memory for a C descriptor. The address of an entity declared using the macro is not usable as an actual argument corresponding to a formal parameter of type CFI_cdesc_t * without an explicit cast. For example, the following code uses CFI_CDESC_T to declare a C descriptor of rank 5 and pass it to CFI_deallocate (18.5.5.4).

```c
CFI_CDESC_T(5) object;
int ind;
... code to define and use C descriptor ...
ind = CFI_deallocate((CFI_cdesc_t *)&object);
```

1 CFI_index_t is a typedef name for a standard signed integer type capable of representing the result of subtracting two pointers.

2 The CFI_MAX_RANK macro has a processor-dependent value equal to the largest rank supported. The value shall be greater than or equal to 15. CFI_rank_t is a typedef name for a standard integer type capable of representing the largest supported rank.

3 The CFI_VERSION macro has a processor-dependent value that encodes the version of the ISO_Fortran_binding.h source file containing this macro. This value should be increased if a new version of the source file is incompatible with the previous version.

4 The macros in Table 18.3 are for use as attribute codes. The values shall be nonnegative and distinct. CFI_attribute_t is a typedef name for a standard integer type capable of representing the values of the attribute codes.

<table>
<thead>
<tr>
<th>Macro name</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFI_attribute_pointer</td>
<td>data pointer</td>
</tr>
<tr>
<td>CFI_attribute_allocatable</td>
<td>allocatable</td>
</tr>
<tr>
<td>CFI_attribute_other</td>
<td>nonallocatable nonpointer</td>
</tr>
</tbody>
</table>

5 The macros in Table 18.4 are for use as type specifiers. The value for CFI_type_other shall be negative and distinct from all other type specifiers. CFI_type_struct specifies a C structure that is interoperable with a Fortran derived type; its value shall be positive and distinct from all other type specifiers. If a C type is not interoperable with a Fortran type and kind supported by the Fortran processor, its macro shall evaluate to a negative value. Otherwise, the value for an intrinsic type shall be positive.

6 If the processor supports interoperability of a Fortran intrinsic type with a C type not listed in Table 18.4, the processor shall define a type specifier value for that type which is positive and distinct from all other type specifiers.

7 CFI_attribute_pointer specifies a data object with the Fortran POINTER attribute. CFI_attribute_allocatable specifies an object with the Fortran ALLOCATABLE attribute. CFI_attribute_other specifies a nonallocatable nonpointer object.

8 The macros in Table 18.4 are for use as type specifiers. The value for CFI_type_other shall be negative and distinct from all other type specifiers. CFI_type_struct specifies a C structure that is interoperable with a Fortran derived type; its value shall be positive and distinct from all other type specifiers. If a C type is not interoperable with a Fortran type and kind supported by the Fortran processor, its macro shall evaluate to a negative value. Otherwise, the value for an intrinsic type shall be positive.

9 If the processor supports interoperability of a Fortran intrinsic type with a C type not listed in Table 18.4, the processor shall define a type specifier value for that type which is positive and distinct from all other type specifiers.

10 CFI_type_t is a typedef name for a standard integer type capable of representing the values for the supported type specifiers.

<table>
<thead>
<tr>
<th>Macro name</th>
<th>C Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFI_type_signed_char</td>
<td>signed char</td>
</tr>
<tr>
<td>CFI_type_short</td>
<td>short int</td>
</tr>
</tbody>
</table>
ISO_Fortran_binding.h macros for type codes (cont.)

<table>
<thead>
<tr>
<th>Macro name</th>
<th>C Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFI_type_int</td>
<td>int</td>
</tr>
<tr>
<td>CFI_type_long</td>
<td>long int</td>
</tr>
<tr>
<td>CFI_type_long_long</td>
<td>long long int</td>
</tr>
<tr>
<td>CFI_type_size_t</td>
<td>size_t</td>
</tr>
<tr>
<td>CFI_type_int8_t</td>
<td>int8_t</td>
</tr>
<tr>
<td>CFI_type_int16_t</td>
<td>int16_t</td>
</tr>
<tr>
<td>CFI_type_int32_t</td>
<td>int32_t</td>
</tr>
<tr>
<td>CFI_type_int64_t</td>
<td>int64_t</td>
</tr>
<tr>
<td>CFI_type_int_least8_t</td>
<td>int_least8_t</td>
</tr>
<tr>
<td>CFI_type_int_least16_t</td>
<td>int_least16_t</td>
</tr>
<tr>
<td>CFI_type_int_least32_t</td>
<td>int_least32_t</td>
</tr>
<tr>
<td>CFI_type_int_least64_t</td>
<td>int_least64_t</td>
</tr>
<tr>
<td>CFI_type_int_fast8_t</td>
<td>int_fast8_t</td>
</tr>
<tr>
<td>CFI_type_int_fast16_t</td>
<td>int_fast16_t</td>
</tr>
<tr>
<td>CFI_type_int_fast32_t</td>
<td>int_fast32_t</td>
</tr>
<tr>
<td>CFI_type_int_fast64_t</td>
<td>int_fast64_t</td>
</tr>
<tr>
<td>CFI_type_intmax_t</td>
<td>intmax_t</td>
</tr>
<tr>
<td>CFI_typeintptr_t</td>
<td>intptr_t</td>
</tr>
<tr>
<td>CFI_type_ptrdiff_t</td>
<td>ptdiff_t</td>
</tr>
<tr>
<td>CFI_type_float</td>
<td>float</td>
</tr>
<tr>
<td>CFI_type_double</td>
<td>double</td>
</tr>
<tr>
<td>CFI_type_long_double</td>
<td>long double</td>
</tr>
<tr>
<td>CFI_type_float_Complex</td>
<td>float _Complex</td>
</tr>
<tr>
<td>CFI_type_double_Complex</td>
<td>double _Complex</td>
</tr>
<tr>
<td>CFI_type_long_double_Complex</td>
<td>long double _Complex</td>
</tr>
<tr>
<td>CFI_type_Bool</td>
<td>_Bool</td>
</tr>
<tr>
<td>CFI_type_char</td>
<td>char</td>
</tr>
<tr>
<td>CFI_type_cptra</td>
<td>void *</td>
</tr>
<tr>
<td>CFI_type_struct</td>
<td>interoperable C structure</td>
</tr>
<tr>
<td>CFI_type_other</td>
<td>Not otherwise specified</td>
</tr>
</tbody>
</table>

NOTE 18.26

The values for different C types can be the same; for example, CFI_type_int and CFI_type_int32_t might have the same value.

11 The macros in Table 18.5 are for use as error codes. The macro CFI_SUCCESS shall be defined to be the integer constant 0. The value of each macro other than CFI_SUCCESS shall be nonzero and shall be different from the values of the other macros specified in this subclause. Error conditions other than those listed in this subclause should be indicated by error codes different from the values of the macros named in this subclause.

12 The values of the macros in Table 18.5 indicate the error condition described.

Table 18.5: ISO_Fortran_binding.h macros for error codes

<table>
<thead>
<tr>
<th>Macro name</th>
<th>Error condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFI_SUCCESS</td>
<td>No error detected.</td>
</tr>
<tr>
<td>CFI_ERROR_BASE_ADDR_NULL</td>
<td>The base address member of a C descriptor is a null pointer in a context that requires a non-null pointer value.</td>
</tr>
<tr>
<td>CFI_ERROR_BASE_ADDR_NOT_NULL</td>
<td>The base address member of a C descriptor is not a null pointer in a context that requires a null pointer value.</td>
</tr>
<tr>
<td>CFI_INVALID_ELEM_LEN</td>
<td>The value supplied for the element length member of a C descriptor is not valid.</td>
</tr>
</tbody>
</table>
ISO_Fortran_binding.h macros for error codes

<table>
<thead>
<tr>
<th>Macro name</th>
<th>Error condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFI_INVALID_RANK</td>
<td>The value supplied for the rank member of a C descriptor is not valid.</td>
</tr>
<tr>
<td>CFI_INVALID_TYPE</td>
<td>The value supplied for the type member of a C descriptor is not valid.</td>
</tr>
<tr>
<td>CFI_INVALID_ATTRIBUTE</td>
<td>The value supplied for the attribute member of a C descriptor is not valid.</td>
</tr>
<tr>
<td>CFI_INVALID_EXTENT</td>
<td>The value supplied for the extent member of a CFI_dim_t structure is not valid.</td>
</tr>
<tr>
<td>CFI_INVALID_DESCRIPTOR</td>
<td>A C descriptor is invalid in some way.</td>
</tr>
<tr>
<td>CFI_ERROR_MEM_ALLOCATION</td>
<td>Memory allocation failed.</td>
</tr>
<tr>
<td>CFI_ERROR_OUT_OF_BOUNDS</td>
<td>A reference is out of bounds.</td>
</tr>
</tbody>
</table>

18.5.5 Functions declared in ISO_Fortran_binding.h

18.5.5.1 Arguments and results of the functions

Some of the functions described in 18.5.5 return an error indicator; this is an integer value that indicates whether an error condition was detected. The value zero indicates that no error condition was detected, and a nonzero value indicates which error condition was detected. Table 18.5 lists standard error conditions and macro names for their corresponding error codes. A processor is permitted to detect other error conditions. If an invocation of a function defined in 18.5.5 could detect more than one error condition and an error condition is detected, which error condition is detected is processor dependent.

Prototypes for these functions, or equivalent macros, are provided in the ISO_Fortran_binding.h file as described in 18.5.5. It is unspecified whether the functions defined by this header are macros or identifiers declared with external linkage. If a macro definition is suppressed in order to access an actual function, the behavior is undefined.

NOTE 18.27
These functions are allowed to be macros to provide extra implementation flexibility. For example, CFI_establish could include the value of CFI_VERSION in the header used to compile the call to CFI_establish as an extra argument of the actual function used to establish the C descriptor.

18.5.5.2 The CFI_address function

Synopsis. C address of an object described by a C descriptor.

```c
void *CFI_address(const CFI_cdesc_t *dv, const CFI_index_t subscripts[]);
```

Formal Parameters.

dv shall be the address of a C descriptor describing the object. The object shall not be an unallocated allocatable variable or a pointer that is not associated.

subscripts shall be a null pointer or the address of an array of type CFI_index_t. If the object is an array, subscripts shall be the address of an array of CFI_index_t with at least \( n \) elements, where \( n \) is the rank of the object. The value of subscripts[\( i \)] shall be within the bounds of dimension \( i \) specified by the dim member of the C descriptor.

Result Value. If the object is an array of rank \( n \), the result is the C address of the element of the object that
the first \( n \) elements of the \texttt{subscripts} argument would specify if used as subscripts. If the object is scalar, the result is its \texttt{C address}.

4 Example. If \( dv \) is the address of a \texttt{C descriptor} for the Fortran array \( A \) declared as

\[
\text{REAL(C_FLOAT) :: A(100, 100)}
\]

the following code calculates the \texttt{C address} of \( A(5, 10) \):

\[
\begin{align*}
\text{CFI\_index\_t subscripts}[2]; \\
\text{float \*address; } \\
\text{subscripts}[0] &= 4; \\
\text{subscripts}[1] &= 9; \\
\text{address} &= (\text{float \*}) \text{CFI\_address}(dv, \text{subscripts}); \\
\end{align*}
\]

18.5.5.3 \texttt{The CFI\_allocate function}

1 Synopsis. Allocate memory for an object described by a \texttt{C descriptor}.

\[
\begin{align*}
\text{int CFI\_allocate(CFI\_cdesc\_t \*dv, const CFI\_index\_t lower\_bounds[],} \\
&\text{const CFI\_index\_t upper\_bounds[], size\_t elem\_len);} \\
\end{align*}
\]

2 Formal Parameters.

\texttt{dv} shall be the address of a \texttt{C descriptor} specifying the rank and type of the object. The \texttt{base\_addr} member of the \texttt{C descriptor} shall be a null pointer. If the type is not a character type, the \texttt{elem\_len} member shall specify the element length. The \texttt{attribute} member shall have a value of \texttt{CFI\_attribute\_allocatable} or \texttt{CFI\_attribute\_pointer}.

\texttt{lower\_bounds} shall be the address of an array with at least \( dv->rank \) elements, if \( dv->rank>0 \).

\texttt{upper\_bounds} shall be the address of an array with at least \( dv->rank \) elements, if \( dv->rank>0 \).

\texttt{elem\_len} If the type specified in the \texttt{C descriptor} type is a Fortran character type, the value of \texttt{elem\_len} shall be the storage size in bytes of an element of the object; otherwise, \texttt{elem\_len} is ignored.

3 Description. Successful execution of \texttt{CFI\_allocate} allocates memory for the object described by the \texttt{C descriptor} with the address \( dv \) using the same mechanism as the Fortran \texttt{ALLOCATE} statement, and assigns the address of that memory to \( dv->base\_addr \). The first \( dv->rank \) elements of the \texttt{lower\_bounds} and \texttt{upper\_bounds} arguments provide the lower and upper Fortran bounds, respectively, for each corresponding dimension of the object. The supplied lower and upper bounds override any current dimension information in the \texttt{C descriptor}. If the rank is zero, the \texttt{lower\_bounds} and \texttt{upper\_bounds} arguments are ignored. If the type specified in the \texttt{C descriptor} is a character type, the supplied element length overrides the current element-length information in the descriptor.

If an error is detected, the \texttt{C descriptor} is not modified.

4 Result Value. The result is an error indicator.

5 Example. If \( dv \) is the address of a \texttt{C descriptor} for the Fortran array \( A \) declared as

\[
\begin{align*}
\text{REAL, ALLOCATABLE :: A(:, :) } \\
\end{align*}
\]

and the array is not allocated, the following code allocates it to be of shape \([100, 500]\):

\[
\begin{align*}
\text{CFI\_index\_t lower}[2], \text{upper}[2]; \\
\text{int ind; } \\
\text{lower}[0] &= 1; \text{lower}[1] &= 1; \\
\text{upper}[0] &= 100; \text{upper}[1] &= 500; \\
\text{ind} &= \text{CFI\_allocate}(dv, \text{lower, upper, 0}); \\
\end{align*}
\]
18.5.4 The CFI_deallocate function

1 Synopsis. Deallocate memory for an object described by a C descriptor.

```c
int CFI_deallocate(CFI_cdesc_t *dv);
```

2 Formal Parameter. *dv* shall be the address of a C descriptor describing the object. It shall have been allocated using the same mechanism as the Fortran `ALLOCATE` statement. If the object is a pointer, it shall be associated with a target satisfying the conditions for successful deallocation by the Fortran `DEALLOCATE` statement (9.7.3).

3 Description. Successful execution of CFI_deallocate deallocates memory for the object using the same mechanism as the Fortran `DEALLOCATE` statement, and the `base_addr` member of the C descriptor becomes a null pointer.

If an error is detected, the C descriptor is not modified.

4 Result Value. The result is an error indicator.

5 Example. If *dv* is the address of a C descriptor for the Fortran array `A` declared as

```fortran
REAL, ALLOCATABLE :: A(:, :)
```

and the array is allocated, the following code deallocates it:

```c
int ind;
ind = CFI_deallocate(dv);
```

18.5.5.5 The CFI_establish function

1 Synopsis. Establish a C descriptor.

```c
int CFI_establish(CFI_cdesc_t *dv, void *base_addr, CFI_attribute_t attribute,
                  CFI_type_t type, size_t elem_len, CFI_rank_t rank,
                  const CFI_index_t extents[]);
```

2 Formal Parameters.

- **dv** shall be the address of a data object large enough to hold a C descriptor of the rank specified by `rank`. It shall not have the same value as either a C formal parameter that corresponds to a Fortran actual argument or a C actual argument that corresponds to a Fortran dummy argument. It shall not be the address of a C descriptor that describes an allocated allocatable object.

- **base_addr** shall be a null pointer or the base address of the object to be described. If it is not a null pointer, it shall be the address of a contiguous storage sequence that is appropriately aligned (ISO/IEC 9899:2011 3.2) for an object of the type specified by `type`.

- **attribute** shall be one of the attribute codes in Table 18.3. If it is CFI_attribute_allocatable, `base_addr` shall be a null pointer.

- **type** shall be one of the type codes in Table 18.4.

- **elem_len** If the type is equal to CFI_type_struct, CFI_type_other, or a Fortran character type code, `elem_len` shall be greater than zero and equal to the storage size in bytes of an element of the object. Otherwise, `type` will be ignored.

- **rank** shall have a value in the range $0 \leq \text{rank} \leq \text{CFI_MAX_RANK}$. It specifies the rank of the object.

- **extents** is ignored if `rank` is equal to zero or if `base_addr` is a null pointer. Otherwise, it shall be the address of an array with `rank` elements; the value of each element shall be nonnegative, and `extents[i]` specifies the extent of dimension $i$ of the object.
Description. Successful execution of CFI_establish updates the object with the address \texttt{dv} to be an established C descriptor for a nonallocatable nonpointer data object of known shape, an unallocated allocatable object, or a data pointer. If \texttt{base_addr} is not a null pointer, it is for a nonallocatable entity that is a scalar or a contiguous array; if the \texttt{attribute} argument has the value CFI_attribute_pointer, the lower bounds of the object described by \texttt{dv} are set to zero. If \texttt{base_addr} is a null pointer, the established C descriptor is for an unallocated allocatable, a disassociated pointer, or is a C descriptor that has the attribute CFI_attribute_other but does not describe a data object. If \texttt{base_addr} is the C address of a Fortran data object, the \texttt{type} and \texttt{elem_len} arguments shall be consistent with the type and type parameters of the Fortran data object. The remaining properties of the object are given by the other arguments.

If an error is detected, the object with the address \texttt{dv} is not modified.

Result Value. The result is an error indicator.

NOTE 18.28

CFI_establish is used to initialize a C descriptor declared in C with CFI_CDESC_T before passing it to any other functions as an actual argument, in order to set the rank, attribute, type and element length.

NOTE 18.29

A C descriptor with attribute CFI_attribute_other and \texttt{base_addr} a null pointer can be used as the argument \texttt{result} in calls to CFI_section or CFI_select_part, which will produce a C descriptor for a nonallocatable nonpointer data object.

Examples.

Case (i): The following code fragment establishes a C descriptor for an unallocated rank-one allocatable array that can be passed to Fortran for allocation there.

```c
CFI_rank_t rank;
CFI_CDESC_T(1) field;
int ind;
rank = 1;
ind = CFI_establish((CFI_cdesc_t *)&field, NULL, CFI_attribute_allocatable,
CFI_type_double, 0, rank, NULL);
```

Case (ii): Given the Fortran type definition

```fortran
TYPE, BIND(C) :: T
REAL(C_DOUBLE) :: X
COMPLEX(C_DOUBLE_COMPLEX) :: Y
END TYPE
```

and a Fortran subprogram that has an assumed-shape dummy argument of type T, the following code fragment creates a descriptor \texttt{a_fortran} for an array of size 100 that can be used as the actual argument in an invocation of the subprogram from C:

```c
typedef struct {double x; double _Complex y;} t;
t a_c[100];
CFI_CDESC_T(1) a_fortran;
int ind;
CFI_index_t extent[1];

extent[0] = 100;
ind = CFI_establish((CFI_cdesc_t *)&a_fortran, a_c, CFI_attribute_other,
CFI_type_struct, sizeof(t), 1, extent);
```
18.5.5.6 The CFI_is_contiguous function

Synopsis. Test contiguity of an array.

```c
int CFI_is_contiguous(const CFI_cdesc_t * dv);
```

Formal Parameter. `dv` shall be the address of a C descriptor describing an array. The `base_addr` member of the C descriptor shall not be a null pointer.

Result Value. The value of the result is 1 if the array described by `dv` is contiguous, and 0 otherwise.

NOTE 18.30 Assumed-size and allocatable arrays are always contiguous, and therefore the result of CFI_is_contiguous on a C descriptor for such an array will be equal to 1.

18.5.5.7 The CFI_section function

Synopsis. Update a C descriptor for an array section for which each element is an element of a given array.

```c
int CFI_section(CFI_cdesc_t *result, const CFI_cdesc_t *source,
    const CFI_index_t lower_bounds[], const CFI_index_t upper_bounds[],
    const CFI_index_t strides[]);
```

Formal Parameters.
- `result` shall be the address of a C descriptor with rank equal to the rank of `source` minus the number of zero strides. The `attribute` member shall have the value CFI_attribute_other or CFI_attribute_pointer. If the value of `result` is the same as either a C formal parameter that corresponds to a Fortran actual argument or a C actual argument that corresponds to a Fortran dummy argument, the `attribute` member shall have the value CFI_attribute_pointer.
- `source` shall be the address of a C descriptor that describes a nonallocatable nonpointer array, an allocated allocatable array, or an associated array pointer. The `elem_len` and `type` members of `source` shall have the same values as the corresponding members of `result`.
- `lower_bounds` shall be a null pointer or the address of an array with at least `source->rank` elements. If it is not a null pointer, and `stride_i` is zero or `(upper_i - lower_bounds[i] + stride_i)/stride_i > 0`, the value of `lower_bounds[i]` shall be within the bounds of dimension `i` of `SOURCE`.
- `upper_bounds` shall be a null pointer or the address of an array with at least `source->rank` elements. If `source` describes an assumed-size array, `upper_bounds` shall not be a null pointer. If it is not a null pointer and `stride_i` is zero or `(upper_bounds[i] - lower_i + stride_i)/stride_i > 0`, the value of `upper_bounds[i]` shall be within the bounds of dimension `i` of `SOURCE`.
- `strides` shall be a null pointer or the address of an array with at least `source->rank` elements.

Successful execution of CFI_section updates the `base_addr` and `dim` members of the C descriptor with the address `result` to describe the array section determined by `source`, `lower_bounds`, `upper_bounds`, and `strides`, as follows.

The array section is equivalent to the Fortran array section `SOURCE(seクトsub_1, sectsub_2, ... sectsub_n)`, where `SOURCE` is the array described by `source`, `n` is the rank of that array, and `sectsub_i` is the subscript `lower_i` if `stride_i` is zero, and the section subscript `lower_i : upper_i ; stride_i` otherwise. The value of `lower_i` is the lower bound of dimension `i` of `SOURCE` if `lower_bounds` is a null pointer and `lower_bounds[i]` otherwise. The value of `upper_i` is the upper bound of dimension `i` of `SOURCE` if `upper_bounds` is a null pointer and `upper_bounds[i]` otherwise. The value of `strides` is 1 if `strides` is a null pointer and `strides[i]` otherwise. If `stride_i` has the value zero, `lower_i` shall have the same value as `upper_i`.

If an error is detected, the C descriptor with the address `result` is not modified.
3 Result Value. The result is an error indicator.

4 Examples.

Case (i): If source is already the address of a C descriptor for the rank-one Fortran array A, the lower bounds of A are equal to 1, and the lower bounds in the C descriptor are equal to 0, the following code fragment establishes a new C descriptor section and updates it to describe the array section A(3:5):

```c
CFI_index_t lower[1], strides[1];
CFI_CDESC_T(1) section;
int ind;
lower[0] = 2;
strides[0] = 5;
ind = CFI_establish((CFI_cdesc_t *)&section, NULL, CFI_attribute_other,
                    CFI_type_float, 0, 1, NULL);
ind = CFI_section((CFI_cdesc_t *)&section, source, lower, NULL, strides);
```

Case (ii): If source is already the address of a C descriptor for a rank-two Fortran assumed-shape array A with lower bounds equal to 1, the following code fragment establishes a C descriptor and updates it to describe the rank-one array section A(:, 42):

```c
CFI_index_t lower[2], upper[2], strides[2];
CFI_CDESC_T(1) section;
int ind;
lower[0] = source->dim[0].lower_bound;
upper[0] = source->dim[0].lower_bound + source->dim[0].extent - 1;
strides[0] = 1;
strides[1] = 0;
ind = CFI_establish((CFI_cdesc_t *)&section, NULL, CFI_attribute_other,
                    CFI_type_float, 0, 1, NULL);
ind = CFI_section((CFI_cdesc_t *)&section, source, lower, upper, strides);
```

18.5.5.8 The CFI_select_part function

1 Synopsis. Update a C descriptor for an array section for which each element is a part of the corresponding element of an array.

```c
int CFI_select_part(CFI_cdesc_t *result, const CFI_cdesc_t *source, size_t displacement,
                    size_t elem_len);
```

2 Formal Parameters.

- **result** shall be the address of a C descriptor; result->rank shall have the same value as source->rank and result->attribute shall have the value CFI_attribute_other or CFI_attribute_pointer. If the address specified by result is the value of a C formal parameter that corresponds to a Fortran actual argument or of a C actual argument that corresponds to a Fortran dummy argument, result->attribute shall have the value CFI_attribute_pointer. The value of result->type specifies the type of the array section.

- **source** shall be the address of a C descriptor for a nonallocatable nonpointer array, an allocated allocatable array, or an associated array pointer.

- **displacement** shall have a value $0 \leq \text{displacement} \leq \text{source->elem_len} - 1$, and the sum of the displacement and the size in bytes of an element of the array section shall be less than or equal to source->elem_len. The address displacement bytes greater than the value of source->base_addr is the base of
the array section and shall be appropriately aligned (ISO/IEC 9899:2011, 3.2) for an object of the
type of the array section.

\[ \text{elem_len} \] shall have a value equal to the storage size in bytes of an element of the array section if \text{result->type}
specifies a Fortran character type; otherwise, \text{elem_len} is ignored.

3 **Description.** Successful execution of \text{CFI_select_part} updates the base_addr, dim, and \text{elem_len} members of
the C descriptor with the address result for an array section for which each element is a part of the corresponding
element of the array described by the C descriptor with the address source. The part shall be a component of a
structure, a substring, or the real or imaginary part of a complex value.

If an error is detected, the C descriptor with the address result is not modified.

4 **Result Value.** The result is an error indicator.

5 **Example.** If source is already the address of a C descriptor for the Fortran array A declared with

\[
\begin{align*}
\text{TYPE, BIND(C)} &:: \text{T} \\
\text{REAL(C_DOUBLE)} &:: \text{X} \\
\text{COMPLEX(C_DOUBLE_COMPLEX)} &:: \text{Y} \\
\text{END TYPE} \\
\text{TYPE(T) A(100)}
\end{align*}
\]

the following code fragment establishes a C descriptor for the array \(A%Y\):

\[
\begin{align*}
\text{typedef struct} & \{ \\
\text{double x; double _Complex y;} \\
\} \text{t}; \\
\text{CFI_CDESC_T(1) component;} \\
\text{CFI_cdesc_t * comp_cdesc = (CFI_cdesc_t *)&component;} \\
\text{CFI_index_t extent[]} = \{ 100 \}; \\
(\text{void})\text{CFI_establish(comp_cdesc, NULL, CFI_attribute_other, CFI_type_double_Complex,} \\
\text{sizeof(double _Complex), 1, extent);} \\
(\text{void})\text{CFI_select_part(comp_cdesc, source, offsetof(t,y), 0);} \\
\end{align*}
\]

18.5.5.9 **The CFI_setpointer function**

1 **Synopsis.** Update a C descriptor for a Fortran pointer to be associated with the whole of a given object or to
be disassociated.

\[
\begin{align*}
\text{int CFI_setpointer(CFI_cdesc_t *result, CFI_cdesc_t *source,} \\
\text{\phantom{int} const CFI_index_t lower_bounds[]);} \\
\end{align*}
\]

2 **Formal Parameters.**

result shall be the address of a C descriptor for a Fortran pointer. It is updated using information from
the source and lower_bounds arguments.

source shall be a null pointer or the address of a C descriptor for a nonallocatable nonpointer data object,
an allocated allocatable object, or a data pointer object. If source is not a null pointer, the corre-
sponding values of the \text{elem_len}, \text{rank}, and \text{type} members shall be the same in the C descriptors
with the addresses source and result.

lower_bounds If source is not a null pointer and source->rank is nonzero, lower_bounds shall be a null pointer
or the address of an array with at least source->rank elements.
Result Value. The result is an error indicator.

Description. Successful execution of CFI_setpointer updates the base_addr and dim members of the C descriptor with the address result as follows:

- if source is a null pointer or the address of a C descriptor for a disassociated pointer, the updated C descriptor describes a disassociated pointer;
- otherwise, the C descriptor with the address result becomes a C descriptor for the object described by the C descriptor with the address source, except that if source->rank is nonzero and lower_bounds is not a null pointer, the lower bounds are replaced by the values of the first source->rank elements of the lower_bounds array.

If an error is detected, the C descriptor with the address result is not modified.

Example. If ptr is already the address of a C descriptor for an array pointer of rank 1, the following code updates it to be a C descriptor for a pointer to the same array with lower bound 0.

```c
CFI_index_t lower_bounds[1];
int ind;
lower_bounds[0] = 0;
ind = CFI_setpointer(ptr, ptr, lower_bounds);
```

18.6 Restrictions on C descriptors

1 A C descriptor shall not be initialized, updated, or copied other than by calling the functions specified in 18.5.5.

2 If the address of a C descriptor is a formal parameter that corresponds to a Fortran actual argument or a C actual argument that corresponds to a Fortran dummy argument,
   - the C descriptor shall not be modified if either the corresponding dummy argument in the Fortran interface has the INTENT (IN) attribute or the C descriptor is for a nonallocatable nonpointer object, and
   - the base_addr member of the C descriptor shall not be accessed before it is given a value if the corresponding dummy argument in the Fortran interface has the POINTER and INTENT (OUT) attributes.

NOTE 18.31 In this context, modification refers to any change to the location or contents of the C descriptor, including establishment and update. The intent of these restrictions is that C descriptors remain intact at all times they are accessible to an active Fortran procedure, so that the Fortran code is not required to copy them.

18.7 Restrictions on formal parameters

1 Within a C function, an allocatable object shall be allocated or deallocated only by execution of the CFI_allocate and CFI_deallocate functions. A Fortran pointer can become associated with a target by execution of the CFI_allocate function.

2 Calling CFI_allocate or CFI_deallocate for a C descriptor changes the allocation status of the Fortran variable it describes.

3 If the address of an object is the value of a formal parameter that corresponds to a nonpointer dummy argument in an interface with the BIND attribute, then
   - if the dummy argument has the INTENT (IN) attribute, the object shall not be defined or become undefined, and...
• if the dummy argument has the INTENT (OUT) attribute, the object shall not be referenced before it is defined.

If a formal parameter that is a pointer to CFI_cdesc_t corresponds to a dummy argument in an interoperable procedure interface, a pointer based on the base_addr in that C descriptor shall not be used to access memory that is not part of the object described by the C descriptor.

### 18.8 Restrictions on lifetimes

1. A C descriptor of, or C pointer to, any part of a Fortran object becomes undefined under the same conditions that the association status of a Fortran pointer associated with that object would become undefined, and any further use of it is undefined behavior (ISO/IEC 9899:2011, 3.4.3).

2. A C descriptor whose address is a formal parameter that corresponds to a Fortran dummy argument becomes undefined on return from a call to the function from Fortran. If the dummy argument does not have either the TARGET or ASYNCHRONOUS attribute, all C pointers to any part of the object described by the C descriptor become undefined on return from the call, and any further use of them is undefined behavior.

3. If the address of a C descriptor is passed as an actual argument to a Fortran procedure, the lifetime (ISO/IEC 9899:2011, 6.2.4) of the C descriptor shall not end before the return from the procedure call. If an object is passed to a Fortran procedure as a nonallocatable, nonpointer dummy argument, its lifetime shall not end before the return from the procedure call.

4. If the lifetime of a C descriptor for an allocatable object that was established by C ends before the program exits, the object shall be unallocated at that time.

5. If a Fortran pointer becomes associated with a data object defined by the companion processor, the association status of the Fortran pointer becomes undefined when the lifetime of that data object ends.

NOTE 18.32

The following example illustrates how a C descriptor becomes undefined upon returning from a call to a C function.

```fortran
REAL, TARGET :: X(1000), B
INTERFACE
  REAL FUNCTION CFUN(ARRAY) BIND(C, NAME="Cfun")
  REAL ARRAY(:)
  END FUNCTION
END INTERFACE
B = CFUN(X)
```

Cfun is a C function. Before or during the invocation of Cfun, the processor will create a C descriptor for the array X. On return from Cfun, that C descriptor will become undefined. In addition, because the dummy argument ARRAY does not have the TARGET or ASYNCHRONOUS attribute, a C pointer whose value was set during execution of Cfun to be the address of any part of X will become undefined.

### 18.9 Interoperation with C global variables

#### 18.9.1 General

1. A C variable whose name has external linkage may interoperate with a common block or with a variable declared in the scope of a module. The common block or variable shall be specified to have the BIND attribute.
At most one variable that is associated with a particular C variable whose name has external linkage is permitted to be declared within all the Fortran program units of a program. A variable shall not be initially defined by more than one processor.

If a common block is specified in a BIND statement, it shall be specified in a BIND statement with the same binding label in each scoping unit in which it is declared. A C variable whose name has external linkage interoperates with a common block that has been specified in a BIND statement 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 does not have to be an associated C entity for a Fortran entity with the BIND attribute.

NOTE 18.33

The following are examples of the usage of the BIND attribute for variables and for a common block. The Fortran variables, C_EXTERN and C2, interoperate with the C variables, c_extern and myVariable, respectively. The Fortran common blocks, COM and SINGLE, interoperate with the C variables, com and single, respectively.

```fortran
MODULE LINK_TO_C_VARS
  USE, INTRINSIC :: ISO_C_BINDING
  INTEGER(C_INT), BIND(C) :: C_EXTERN
  INTEGER(C_LONG) :: C2
  BIND(C, NAME='myVariable') :: C2
  COMMON /COM/ R, S
  REAL(C_FLOAT) :: R, S, T
  BIND(C) :: /COM/, /SINGLE/
  COMMON /SINGLE/ T
END MODULE LINK_TO_C_VARS
```

/* Global variables. */
int c_extern;
long myVariable;
struct { float r, s; } com;
float single;

### 18.9.2 Binding labels for common blocks and variables

1. The binding label of a variable or common block is a default character value that specifies the name by which the variable or common block is known to the companion processor.

2. If a variable or common block has the BIND attribute with the NAME= specifier and the value of its expression, after discarding leading and trailing blanks, has nonzero length, the variable or common block has this as its binding label. The case of letters in the binding label is significant. If a variable or common block 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. Otherwise, the variable or common block has no binding label.

3. The binding label of a C variable whose name has external linkage is the same as the name of the C variable. A Fortran variable or common block with the BIND attribute that has the same binding label as a C variable whose name has external linkage is linkage associated (19.5.1.5) with that variable.
18.10 Interoperation with C functions

18.10.1 Definition and reference of interoperable procedures

1 A procedure that is interoperable may be defined either by means other than Fortran or by means of a Fortran
subprogram, but not both. A C function that has an inline definition and no external definition is not considered
to be defined in this sense.

2 If the procedure is defined by means other than Fortran,
   • it shall be describable by a C prototype that is interoperable with the interface, and
   • if it is accessed using its binding label, it shall
     – have a name that has external linkage as defined by 6.2.2 of ISO/IEC 9899:2011, and
     – have the same binding label as the interface.

3 A reference to such a procedure causes the function described by the C prototype to be called as specified in

4 A reference in C to a procedure that has the BIND attribute, has the same binding label, and is defined by means
of Fortran, causes the Fortran procedure to be invoked. A C function shall not invoke a function pointer whose
value is the result of a reference to C_FUNLOC with a noninteroperable argument.

5 A procedure defined by means of Fortran shall not invoke setjmp or longjmp (ISO/IEC 9899:2011, 7.13). If a
procedure defined by means other than Fortran invokes setjmp or longjmp, that procedure shall not cause any
procedure defined by means of Fortran to be invoked. A procedure defined by means of Fortran shall not be

6 If a procedure defined by means of Fortran and a procedure defined by means other than Fortran perform
input/output operations on the same external file, the results are processor dependent (12.5.4).

7 If the value of a C function pointer will be the result of a reference to C_FUNLOC with a noninteroperable
argument, it is recommended that the C function pointer be declared to have the type void (*)( ).

18.10.2 Binding labels for procedures

1 The binding label of a procedure is a default character value that specifies the name by which a procedure with
the BIND attribute is known to the companion processor.

2 If a procedure has the BIND attribute with the NAME= specifier and the value of its expression, after discarding
leading and trailing blanks, has nonzero length, the procedure has this as its binding label. The case of letters
in the binding label is significant. If a procedure has the BIND attribute with no NAME= specifier, and the
procedure is not a dummy procedure, internal procedure, or procedure pointer, then the binding label of the
procedure is the same as the name of the procedure using lower case letters. Otherwise, the procedure has no
binding label.

C1807 A procedure defined in a submodule shall not have a binding label unless its interface is declared in the
ancestor module.

3 The binding label for a C function whose name has external linkage is the same as the C function name.

NOTE 18.34
In the following sample, the binding label of C_SUB is "c_sub", and the binding label of C_FUNC is
"C_func".

SUBROUTINE C_SUB() BIND(C)
    ...

NOTE 18.34 (cont.)

END SUBROUTINE C_SUB

INTEGER(C_INT) FUNCTION C_FUNC() BIND(C, NAME="C_func")
  USE, INTRINSIC :: ISO_C_BINDING
  ...
END FUNCTION C_FUNC

ISO/IEC 9899:2011 permits functions to have names that are not permitted as Fortran names; it also
distinguishes between names that would be considered as the same name in Fortran. For example, a C
name can begin with an underscore, and C names that differ in case are distinct names.

The specification of a binding label allows a program to use a Fortran name to refer to a procedure defined
by a companion processor.

18.10.3 Exceptions and IEEE arithmetic procedures

1 A procedure defined by means other than Fortran shall not use signal (ISO/IEC 9899:2011, 7.14.1) to change the
handling of any exception that is being handled by the Fortran processor.

2 A procedure defined by means other than Fortran shall not alter the floating-point status (17.7) other than by
setting an exception flag to signaling.

3 The values of the floating-point exception flags on entry to a procedure defined by means other than Fortran are
processor dependent.

18.10.4 Asynchronous communication

1 Asynchronous communication for a Fortran variable with the ASYNCHRONOUS attribute occurs through the
action of procedures defined by means other than Fortran. It is initiated by execution of an asynchronous
communication initiation procedure and completed by execution of an asynchronous communication completion
procedure. Between the execution of the initiation and completion procedures, any variable of which any part
is associated with any part of the asynchronous communication variable is a pending communication affector.
Whether a procedure is an asynchronous communication initiation or completion procedure is processor depend-
ent.

2 Asynchronous communication is either input communication or output communication. For input communication,
a pending communication affector shall not be referenced, become defined, become undefined, become associated
with a dummy argument that has the VALUE attribute, or have its pointer association status changed. For
output communication, a pending communication affector shall not be redefined, become undefined, or have its
pointer association status changed. The restrictions for asynchronous input communication are the same as for
asynchronous input data transfer. The restrictions for asynchronous output communication are the same as for
asynchronous output data transfer.

NOTE 18.35
Asynchronous communication can be used for nonblocking MPI calls such as MPI_IRECV and MPI_-
ISEND. For example,

```fortran
REAL :: BUF(100,100)
... ! Code that involves BUF
BLOCK
  ASYNCHRONOUS :: BUF
  CALL MPI_IRECV(BUF,...REQ,...)
... ! Code that does not involve BUF
```
NOTE 18.35 (cont.)

CALL MPI_WAIT(REQ,...)
END BLOCK
... ! Code that involves BUF

In this example, there is asynchronous input communication and BUF is a pending communication affector between the two calls. MPI_IRECV can return while the communication (reading values into BUF) is still underway. The intent is that the code between MPI_IRECV and MPI_WAIT can execute without waiting for this communication to complete.

Similar code with the call of MPI_IRECV replaced by a call of MPI_ISEND is asynchronous output communication.
19 Scope, association, and definition

19.1 Scopes, identifiers, and entities

1 An entity is identified by an identifier.

2 The scope of
   • a global identifier is a program (5.2.2),
   • a local identifier is an inclusive scope,
   • an identifier of a construct entity is that construct (10.2.4, 11.1), and
   • an identifier of a statement entity is that statement or part of that statement (6.3),

excluding any nested scope where the identifier is treated as the identifier of a different entity (19.3, 19.4).

3 An entity may be identified by
   • an image index (3.85),
   • a name (3.103),
   • a statement label (3.135),
   • an external input/output unit number (12.5),
   • an identifier of a pending data transfer operation (12.6.2.9, 12.7),
   • a submodule identifier (14.2.3),
   • a generic identifier (3.79), or
   • a binding label (3.15).

4 By means of association, an entity may be referred to by the same identifier or a different identifier in a different
   scope, or by a different identifier in the same scope.

19.2 Global identifiers

1 Program units, common blocks, external procedures, entities with binding labels, external input/output units,
   pending data transfer operations, and images are global entities of a program. The name of a common block with
   no binding label, external procedure with no binding label, or program unit that is not a submodule is a global
   identifier. The submodule identifier of a submodule is a global identifier. A binding label of an entity of the
   program is a global identifier. An entity of the program shall not be identified by more than one binding label.

2 The global identifier of an entity shall not be the same as the global identifier of any other entity. Furthermore, a
   binding label shall not be the same as the global identifier of any other global entity, ignoring differences in case.
   A processor may assign a global identifier to an entity that is not specified by this document to have a global
   identifier (such as an intrinsic procedure); in such a case, the processor shall ensure that this assigned global
   identifier differs from all other global identifiers in the program.

NOTE 19.1
An intrinsic module is not a program unit, so a global identifier can be the same as the name of an intrinsic
module.
NOTE 19.2
Submodule identifiers are global identifiers, but because they consist of a module name and a descendant submodule name, the name of a submodule can be the same as the name of another submodule so long as they do not have the same ancestor module.

19.3 Local identifiers

19.3.1 Classes of local identifiers

1 Identifiers of entities in the classes

   (1) except for statement or construct entities (19.4), named variables, named constants, named procedure pointers, named constructs, statement functions, internal procedures, module procedures, dummy procedures, intrinsic procedures, external procedures that have binding labels, intrinsic modules, abstract interfaces, generic interfaces, derived types, namelist groups, external procedures accessed via USE, and statement labels,

   (2) type parameters, components, and type-bound procedure bindings, in a separate class for each type,

   (3) argument keywords, in a separate class for each procedure with an explicit interface, and

   (4) common blocks that have binding labels

are local identifiers.

2 Within its scope, a local identifier of an entity of class (1) or class (4) shall not be the same as a global identifier used in that scope unless the global identifier

   • is used only as the use-name of a rename in a USE statement,

   • is a common block name (19.3.2),

   • is an external procedure name that is also a generic name, or

   • is an external function name and the inclusive scope is its defining subprogram (19.3.3).

3 Within its scope, a local identifier of one class shall not be the same as another local identifier of the same class, except that a generic name may be the same as the name of a procedure as explained in 15.4.3.4 or the same as the name of a derived type (7.5.10). A local identifier of one class may be the same as a local identifier of another class.

NOTE 19.3
An intrinsic procedure is inaccessible by its own name in a scoping unit that uses the same name as a local identifier of class (1) for a different entity. For example, in the program fragment

SUBROUTINE SUB

    ... A = SIN (K) ...

END SUBROUTINE SUB

any reference to function SIN in subroutine SUB refers to the internal function SIN, not to the intrinsic function of the same name.
A local identifier identifies an entity in a scope and may be used to identify an entity in another scope except in the following cases.

- The name that appears as a subroutine-name in a subroutine-stmt has limited use within the scope established by the subroutine-stmt. It can be used to identify recursive references of the subroutine or to identify a common block (the latter is possible only for internal and module subroutines).
- The name that appears as a function-name in a function-stmt has limited use within the scope established by that function-stmt. It can be used to identify the function result, to identify recursive references of the function, or to identify a common block (the latter is possible only for internal and module functions).
- The name that appears as an entry-name in an entry-stmt has limited use within the scope of the subprogram in which the entry-stmt appears. It can be used to identify the function result if the subprogram is a function, to identify recursive references, or to identify a common block (the latter is possible only if the entry-stmt is in a module subprogram).

19.3.2 Local identifiers that are the same as common block names

A name that identifies a common block in a scoping unit shall not be used to identify a constant or an intrinsic procedure in that scoping unit. If a local identifier of class (1) is also the name of a common block, the appearance of that name in any context other than as a common block name in a BIND, COMMON, or SAVE statement is an appearance of the local identifier.

19.3.3 Function results

For each FUNCTION statement or ENTRY statement in a function subprogram, there is a function result. If there is no RESULT clause, the function result has the same name as the function being defined; otherwise, the function result has the name specified in the RESULT clause.

19.3.4 Components, type parameters, and bindings

A component name has the scope of its derived-type definition. Outside the type definition, it may also appear within a designator of a component of a structure of that type or as a component keyword in a structure constructor for that type.

A type parameter name has the scope of its derived-type definition. Outside the derived-type definition, it may also appear as a type parameter keyword in a derived-type-spec for the type or as the type-param-name of a type-param-inquiry.

The binding name (7.5.5) of a type-bound procedure has the scope of its derived-type definition. Outside of the derived-type definition, it may also appear as the binding-name in a procedure reference.

A generic binding for which the generic-spec is not a generic-name has a scope that consists of all scoping units in which an entity of the type is accessible.

A component name or binding name may appear only in a scope in which it is accessible.

The accessibility of components and bindings is specified in 7.5.4.8 and 7.5.5.

19.3.5 Argument keywords

As an argument keyword, a dummy argument name in an internal procedure, module procedure, or an interface body has a scope of the scoping unit of the host of the procedure or interface body. As an argument keyword, the name of a dummy argument of a procedure declared by a procedure declaration statement that specifies an explicit interface has a scope of the scoping unit containing the procedure declaration statement. It may appear only in a procedure reference for the procedure of which it is a dummy argument. If the procedure is accessible...
in another scoping unit by use or host association (19.5.1.3, 19.5.1.4), the argument keyword is accessible for procedure references for that procedure in that scoping unit.

A dummy argument name in an intrinsic procedure has a scope as an argument keyword of the scoping unit in which the reference to the procedure occurs. As an argument keyword, it may appear only in a procedure reference for the procedure of which it is a dummy argument.

19.4 Statement and construct entities

A variable that appears as a data-i-do-variable in a DATA statement or an ac-do-variable in an array constructor, as a dummy argument in a statement function statement, or as an index-name in a FORALL statement is a statement entity. A variable that appears as an index-name in a FORALL or DO CONCURRENT, as an associate-name in a SELECT TYPE or ASSOCIATE construct, or as a coarray-name in a codimension-decl in a CHANGE TEAM construct is a construct entity. A variable that has LOCAL or LOCAL_INIT locality in a DO CONCURRENT construct is a construct entity. An entity that is explicitly declared in the specification part of a BLOCK construct, other than only in ASYNCHRONOUS and VOLATILE statements, is a construct entity. Two construct entities of the same construct shall not have the same identifier.

Even if the name of a statement entity is the same as another identifier and the statement is in the scope of that identifier, within the scope of the statement entity the name is interpreted as that of the statement entity.

The name of a statement entity shall not be the same as an accessible global identifier or local identifier of class (1) (19.3.1), except for a common block name or a scalar variable name. Within the scope of a statement entity, another statement entity shall not have the same name.

The name of a data-i-do-variable in a DATA statement or an ac-do-variable in an array constructor has a scope of its data-implied-do or ac-implied-do. It is a scalar variable. If integer-type-spec appears in data-implied-do or ac-implied-do-control it has the specified type and type parameters; otherwise it has the type and type parameters that it would have if it were the name of a variable in the innermost executable construct or scoping unit that includes the DATA statement or array constructor, and this type shall be integer type. It has no other attributes. The appearance of a name as a data-i-do-variable of an implied DO in a DATA statement or an ac-do-variable in an array constructor is not an implicit declaration of a variable whose scope is the scoping unit that contains the statement.

The name of a variable that appears as an index-name in a DO CONCURRENT construct, FORALL statement, or FORALL construct has a scope of the statement or construct. It is a scalar variable. If integer-type-spec appears in concurrent-header it has the specified type and type parameters; otherwise it has the type and type parameters that it would have if it were the name of a variable in the innermost executable construct or scoping unit that includes the DO CONCURRENT or FORALL, and this type shall be integer type. It has no other attributes. The appearance of a name as an index-name in a DO CONCURRENT construct, FORALL statement, or FORALL construct is not an implicit declaration of a variable whose scope is the scoping unit that contains the statement or construct.

A variable that has LOCAL or LOCAL_INIT locality in a DO CONCURRENT construct has the scope of that construct. Its attributes are specified in 11.1.7.5.

If integer-type-spec does not appear in a concurrent-header, an index-name shall not be the same as an accessible global identifier, local identifier, or identifier of an outer construct entity, except for a common block name or
a scalar variable name. An *index-name* of a contained DO CONCURRENT construct, FORALL statement, or FORALL construct shall not be the same as an *index-name* of any of its containing DO CONCURRENT or FORALL constructs.

The *associate name* of a SELECT TYPE construct has a separate scope for each block of the construct. Within each block, it has the declared type, dynamic type, type parameters, rank, and bounds specified in 11.1.11.2.

The *associate names* of an ASSOCIATE construct have the scope of the block. They have the declared type, dynamic type, type parameters, rank, and bounds specified in 11.1.3.2.

The *associate names* of a CHANGE TEAM construct have the scope of the block. They have the declared type, dynamic type, type parameters, rank, corank, bounds, and cobounds specified in 11.1.5.

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 is a scalar that 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; it has no other attributes.

### 19.5 Association

#### 19.5.1 Name association

##### 19.5.1.1 Forms of name association

There are five forms of name association: argument association, use association, host association, linkage association, and construct association. Argument, use, and host association provide mechanisms by which entities known in one scope may be accessed in another scope.

##### 19.5.1.2 Argument association

The rules governing argument association are given in Clause 15. As explained in 15.5, execution of a procedure reference establishes a correspondence between each actual argument and a dummy argument and thus an association between each present dummy argument and its effective argument. Argument association can be sequence association (15.5.2.11).

The name of the dummy argument may be different from the name, if any, of its effective argument. The dummy argument name is the name by which the effective argument is known, and by which it may be accessed, in the referenced procedure.

**NOTE 19.4**

An effective argument can be a nameless data entity, such as the result of evaluating an expression that is not simply a variable or constant.

Upon termination of execution of a procedure reference, all argument associations established by that reference are terminated. A dummy argument of that procedure can be associated with an entirely different effective argument in a subsequent invocation of the procedure.
### 19.5.1.3 Use association

Use association is the association of names in different scopes specified by a USE statement. The rules governing use association are given in 14.2.2. They allow for renaming of entities being accessed. Use association allows access in one scope to entities defined or declared in another scope; it remains in effect throughout the execution of the program.

### 19.5.1.4 Host association

A nested scoping unit has access to named entities from its host as specified in 8.8. In the case of an internal subprogram, the access is to the entities in its host instance. The accessed entities are identified by the same identifier and have the same attributes as in the host, except that a local entity may have the ASYNCHRONOUS attribute even if the host entity does not, and a noncoarray local entity may have the VOLATILE attribute even if the host entity does not. The accessed entities are named data objects, derived types, abstract interfaces, procedures, generic identifiers, and namelist groups.

If an entity that is accessed by use association has the same nongeneric name as a host entity, the host entity is inaccessible by that name. The name of an external procedure that is given the EXTERNAL attribute (8.5.9) within the scoping unit, or a name that appears within the scoping unit as a module-name in a use-stmt is a global identifier; any entity of the host that has this as its nongeneric name is inaccessible by that name. A name that appears in the scoping unit as

1. a function-name in a stmt-function-stmt or in an entity-decl in a type-declaration-stmt, unless it is a global identifier,
2. an object-name in an entity-decl in a type-declaration-stmt, in a pointer-stmt, in a save-stmt, in an allocatable-stmt, or in a target-stmt,
3. a type-param-name in a derived-type-stmt,
4. a named-constant in a named-constant-def in a parameter-stmt,
5. a coarray-name in a codimension-stmt,
6. an array-name in a dimension-stmt,
7. a variable-name in a common-block-object in a common-stmt,
8. a procedure pointer given the EXTERNAL attribute in the scoping unit,
9. the name of a variable that is wholly or partially initialized in a data-stmt,
10. the name of an object that is wholly or partially equivalenced in an equivalence-stmt,
11. a dummy-arg-name in a function-stmt, in a subroutine-stmt, in an entry-stmt, or in a stmt-function-stmt,
12. a result-name in a function-stmt or in an entry-stmt,
13. the name of an entity declared by an interface body, unless it is a global identifier,
14. an intrinsic-procedure-name in an intrinsic-stmt,
15. a namelist-group-name in a namelist-stmt,
16. a generic-name in a generic-spec in an interface-stmt, or
17. the name of a named construct

is a local identifier in the scoping unit and any entity of the host that has this as its nongeneric name is inaccessible by that name by host association. If a scoping unit is the host of a derived-type definition or a subprogram that does not define a separate module procedure, the name of the derived type or of any procedure defined by the subprogram is a local identifier in the scoping unit; any entity of the host that has this as its nongeneric name is
inaccessible by that name. Local identifiers of a subprogram are not accessible to its host.

**NOTE 19.5**

A name that appears in an ASYNCHRONOUS or VOLATILE statement is not necessarily the name of a local variable. In an internal or module procedure, if a variable that is accessible via host association is specified in an ASYNCHRONOUS or VOLATILE statement, that host variable is given the ASYNCHRONOUS or VOLATILE attribute in the local scope.

If a host entity is inaccessible only because a local variable with the same name is wholly or partially initialized in a DATA statement, the local variable shall not be referenced or defined prior to the DATA statement.

If a derived-type name of a host is inaccessible, data entities of that type or subobjects of such data entities still can be accessible.

**NOTE 19.6**

An interface body that is not a module procedure interface body accesses by host association only those entities made accessible by IMPORT statements.

If an external or dummy procedure with an implicit interface is accessed via host association, then it shall have the EXTERNAL attribute in the host scoping unit; if it is invoked as a function in the inner scoping unit, its type and type parameters shall be established in the host scoping unit. The type and type parameters of a function with the EXTERNAL attribute are established in a scoping unit if that scoping unit explicitly declares them, invokes the function, accesses the function from a module, or accesses the function from its host where its type and type parameters are established.

If an intrinsic procedure is accessed via host association, then it shall be established to be intrinsic in the host scoping unit. An intrinsic procedure is established to be intrinsic in a scoping unit if that scoping unit explicitly gives it the INTRINSIC attribute, invokes it as an intrinsic procedure, accesses it from a module, or accesses it from its host where it is established to be intrinsic.

**NOTE 19.7**

A host subprogram and an internal subprogram can contain the same and differing use-associated entities, as illustrated in the following example.

```plaintext
MODULE B; REAL BX, Q; INTEGER IX, JX; END MODULE B
MODULE C; REAL CX; END MODULE C
MODULE D; REAL DX, DY, DZ; END MODULE D
MODULE E; REAL EX, EY, EZ; END MODULE E
MODULE F; REAL FX; END MODULE F
MODULE G; USE F; REAL GX; END MODULE G
PROGRAM A
USE B; USE C; USE D
... CONTAINS
  SUBROUTINE INNER_PROC (Q)
    USE C        ! Not needed
    USE B, ONLY: BX  ! Entities accessible are BX, IX, and JX
    ! if no other IX or JX
```

ISO/IEC JTC 1/SC 22/WG5/N2118
NOTE 19.7 (cont.)

```
! is accessible to INNER_PROC
! Q is local to INNER_PROC,
! because Q is a dummy argument
USE D, X => DX ! Entities accessible are DX, DY, and DZ
! X is local name for DX in INNER_PROC
! X and DX denote same entity if no other
! entity DX is local to INNER_PROC
USE E, ONLY: EX ! EX is accessible in INNER_PROC, not in program A
! EY and EZ are not accessible in INNER_PROC
! or in program A
USE G ! FX and GX are accessible in INNER_PROC
...
END SUBROUTINE INNER_PROC
END PROGRAM A
```

Because program A contains the statement

```
USE B
```

all of the entities in module B, except for Q, are accessible in INNER_PROC, even though INNER_PROC contains the statement

```
USE B, ONLY: BX
```

The USE statement with the ONLY option means that this particular statement brings in only the entity named, not that this is the only variable from the module accessible in this scoping unit.

**NOTE 19.8**

For more examples of host association, see subclause C.13.

### 19.5.1.5 Linkage association

Linkage association occurs between a module variable that has the BIND attribute and the C variable with which it interoperates, or between a Fortran common block and the C variable with which it interoperates (18.9). Such association remains in effect throughout the execution of the program.

### 19.5.1.6 Construct association

Execution of a SELECT RANK or SELECT TYPE statement establishes an association between the selector and the associate name of the construct. Execution of an ASSOCIATE or CHANGE TEAM statement establishes an association between each selector and the corresponding associate name of the construct.

In an ASSOCIATE or SELECT TYPE construct, the following rules apply:

- If a selector is allocatable, it shall be allocated; the associate name is associated with the data object and does not have the ALLOCATABLE attribute.
- If a selector has the POINTER attribute, it shall be associated; the associate name is associated with the target of the pointer and does not have the POINTER attribute.
If the selector is a variable other than an array section having a vector subscript, the association is with the data object specified by the selector; otherwise, the association is with the value of the selector expression, which is evaluated prior to execution of the block.

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. On completion of execution of the construct, the association is terminated.

NOTE 19.9
The association between the associate name and a data object is established prior to execution of the block and is not affected by subsequent changes to variables that were used in subscripts or substring ranges in the selector.

19.5.2 Pointer association

19.5.2.1 General

1 Pointer association between a pointer and a target allows the target to be referenced by a reference to the pointer. At different times during the execution of a program, a pointer may be undefined, associated with different targets on its own image, or be disassociated. If a pointer is associated with a target, the definition status of the pointer is either defined or undefined, depending on the definition status of the target. If the pointer has deferred type parameters or shape, their values are assumed from the target. If the pointer is polymorphic, its dynamic type is assumed from the dynamic type of the target.

19.5.2.2 Pointer association status

1 A pointer may have a pointer association status of associated, disassociated, or undefined. Its association status may change during execution of a program. Unless a pointer is initialized (explicitly or by default), it has an initial association status of undefined. A pointer may be initialized to have an association status of disassociated or associated.

NOTE 19.10
A pointer from a module program unit might be accessible in a subprogram via use association. Such pointers have a lifetime that is greater than targets that are declared in the subprogram, unless such targets are saved. Therefore, if such a pointer is associated with a local target, there is the possibility that when a procedure defined by the subprogram completes execution, the target will cease to exist, leaving the pointer “dangling”. This document considers such pointers to have an undefined association status. They are neither associated nor disassociated. They cannot be used again in the program until their status has been reestablished. A processor is not required to detect when a pointer target ceases to exist.

19.5.2.3 Events that cause pointers to become associated

1 A pointer becomes associated when any of the following events occur.

(1) The pointer is allocated (9.7.1) as the result of the successful execution of an ALLOCATE statement referencing the pointer.

(2) The pointer is pointer-assigned to a target (10.2.2) that is associated or is specified with the TARGET attribute and, if allocatable, is allocated.
(3) The pointer is a subobject of an object that is allocated by an ALLOCATE statement in which SOURCE= appears and the corresponding subobject of source-expr is associated.

(4) The pointer is a dummy argument and its corresponding actual argument is not a pointer.

(5) The pointer is a default-initialized subcomponent of an object, the corresponding initializer is not a reference to the intrinsic function NULL, and

(a) a procedure is invoked with this object as an actual argument corresponding to a nonpointer nonallocatable dummy argument with INTENT (OUT),

(b) a procedure with this object as an unsaved nonpointer nonallocatable local variable is invoked,

(c) a BLOCK construct is entered and this object is an unsaved local nonpointer nonallocatable local variable of the BLOCK construct,

or

(d) this object is allocated other than by an ALLOCATE statement in which SOURCE= appears.

19.5.2.4 Events that cause pointers to become disassociated

1 A pointer becomes disassociated when

(1) the pointer is nullified (9.7.2),

(2) the pointer is deallocated (9.7.3),

(3) the pointer is pointer-assigned (10.2.2) to a disassociated pointer,

(4) the pointer is a subobject of an object that is allocated by an ALLOCATE statement in which SOURCE= appears and the corresponding subobject of source-expr is disassociated,

or

(5) the pointer is a default-initialized subcomponent of an object, the corresponding initializer is a reference to the intrinsic function NULL, and

(a) a procedure is invoked with this object as an actual argument corresponding to a nonpointer nonallocatable dummy argument with INTENT (OUT),

(b) a procedure with this object as an unsaved nonpointer nonallocatable local variable is invoked,

(c) a BLOCK construct is entered and this object is an unsaved local nonpointer nonallocatable local variable of the BLOCK construct,

or

(d) this object is allocated other than by an ALLOCATE statement in which SOURCE= appears.

19.5.2.5 Events that cause the association status of pointers to become undefined

1 The association status of a pointer becomes undefined when

(1) the pointer is pointer-assigned to a target that has an undefined association status,

(2) the pointer is pointer-assigned to a target on a different image,

(3) the target of the pointer is deallocated other than through the pointer,

(4) the target of the pointer is a data object defined by the companion processor and the lifetime of that data object ends,

(5) the allocation transfer procedure (16.9.137) is executed, the pointer is associated with the argument FROM, and the argument TO does not have the TARGET attribute,
(6) completion of execution of an instance of a subprogram causes the pointer’s target to become undefined (item (3) of 19.6.6),

(7) completion of execution of a BLOCK construct causes the pointer’s target to become undefined (item (23) of 19.6.6),

(8) execution of the host instance of a procedure pointer is completed,

(9) execution of an instance of a subprogram completes and the pointer is declared or accessed in the subprogram that defines the procedure if the pointer
   (a) does not have the SAVE attribute,
   (b) is not in blank common,
   (c) is not in a named common block that is declared in any other scoping unit that is in execution,
   (d) is not accessed by host association, and
   (e) is not the result of a function declared to have the POINTER attribute,

(10) execution of an instance of a subprogram completes, the pointer is associated with a dummy argument of the procedure, and
   (a) the effective argument does not have the TARGET attribute or is an array section with a vector subscript, or
   (b) the dummy argument has the VALUE attribute,

(11) a BLOCK construct completes execution and the pointer is an unsaved construct entity of that BLOCK construct,

(12) a DO CONCURRENT construct is terminated and the pointer’s association status was changed in more than one iteration of the construct,

(13) an iteration of a DO CONCURRENT construct completes and the pointer is associated with a variable of that construct that has LOCAL or LOCAL_INIT locality,

(14) the pointer is a subcomponent of an object that is allocated and either
   (a) the pointer is not default-initialized and SOURCE= does not appear, or
   (b) SOURCE= appears and the association status of the corresponding subcomponent of source-expr is undefined,

(15) the pointer is a subcomponent of an object, the pointer is not default-initialized, and a procedure is invoked with this object as an actual argument corresponding to a dummy argument with INTENT (OUT),

(16) a procedure is invoked with the pointer as an actual argument corresponding to a pointer dummy argument with INTENT (OUT), or

(17) evaluation of an expression containing a function reference that need not be evaluated completes, if execution of that function would change the association status of the pointer.

19.5.2.6 Other events that change the association status of pointers

1 When a pointer becomes associated with another pointer by argument association, construct association, or host association, the effects on its association status are specified in 19.5.5.

2 While two pointers are name associated, storage associated, or inheritance associated, if the association status of one pointer changes, the association status of the other changes accordingly.
3 The association status of a pointer object with the VOLATILE attribute might change by means not specified
by the program.

19.5.2.7 Pointer definition status

1 The definition status of an associated pointer is that of its target. If a pointer is associated with a definable
target, it may be defined or become undefined according to the rules for a variable (19.6). The definition status
of a pointer that is not associated is undefined.

19.5.2.8 Relationship between association status and definition status

1 If the association status of a pointer is disassociated or undefined, the pointer shall not be referenced or dealloc-
ated. Whatever its association status, a pointer always may be nullified, allocated, or pointer-assigned. A nullified
pointer is disassociated. When a pointer is allocated, it becomes associated but undefined. When a pointer is
pointer-assigned, its association and definition status become those of the specified data-target or proc-target.

19.5.3 Storage association

19.5.3.1 General

Storage sequences are used to describe relationships that exist among variables and common blocks. Storage asso-
ciation is the association of two or more data objects that occurs when two or more storage sequences share or
are aligned with one or more storage units.

19.5.3.2 Storage sequence

A storage sequence is a sequence of storage units. The size of a storage sequence is the number of storage units
in the storage sequence. A storage unit is a character storage unit, a numeric storage unit, a file storage unit
(12.3.5), or an unspecified storage unit. The sizes of the numeric storage unit, the character storage unit and the
file storage unit are the values of constants in the ISO_FORTRAN_ENV intrinsic module (16.10.2).

2 In a storage association context

(1) a nonpointer scalar object that is default integer, default real, or default logical occupies a single
numeric storage unit,
(2) a nonpointer scalar object that is double precision real or default complex occupies two contiguous
numeric storage units,
(3) a default character nonpointer scalar object of character length len occupies len contiguous character
storage units,
(4) if C character kind is not the same as default character kind a nonpointer scalar object of type char-
acter with the C character kind (18.2.2) and character length len occupies len contiguous unspecified
storage units,
(5) a nonpointer scalar object of sequence type occupies a sequence of storage sequences corresponding
to the sequence of its ultimate components,
(6) a nonpointer scalar object of any type not specified in items (1)-(5) occupies a single unspecified
storage unit that is different for each case and each set of type parameter values, and that is different
from the unspecified storage units of item (4),
(7) a nonpointer array occupies a sequence of contiguous storage sequences, one for each array element,
in array element order (9.5.3.2), and
A data pointer occupies a single unspecified storage unit that is different from that of any nonpointer object and is different for each combination of type, type parameters, and rank. A data pointer that has the CONTIGUOUS attribute occupies a storage unit that is different from that of a data pointer that does not have the CONTIGUOUS attribute.

A sequence of storage sequences forms a storage sequence. The order of the storage units in such a composite storage sequence is that of the individual storage units in each of the constituent storage sequences taken in succession, ignoring any zero-sized constituent sequences.

Each common block has a storage sequence (8.10.2.2).

19.5.3.3 Association of storage sequences

Two nonzero-sized storage sequences $s_1$ and $s_2$ are storage associated if the $i$th storage unit of $s_1$ is the same as the $j$th storage unit of $s_2$. This causes the $(i + k)$th storage unit of $s_1$ to be the same as the $(j + k)$th storage unit of $s_2$, for each integer $k$ such that $1 \leq i + k \leq \text{size of } s_1$ and $1 \leq j + k \leq \text{size of } s_2$ where size of measures the number of storage units.

Storage association also is defined between two zero-sized storage sequences, and between a zero-sized storage sequence and a storage unit. A zero-sized storage sequence in a sequence of storage sequences is storage associated with its successor, if any. If the successor is another zero-sized storage sequence, the two sequences are storage associated. If the successor is a nonzero-sized storage sequence, the zero-sized sequence is storage associated with the first storage unit of the successor. Two storage units that are each storage associated with the same zero-sized storage sequence are the same storage unit.

19.5.3.4 Association of scalar data objects

Two scalar data objects are storage associated if their storage sequences are storage associated. Two scalar entities are totally associated if they have the same storage sequence. Two scalar entities are partially associated if they are associated without being totally associated.

The definition status and value of a data object affects the definition status and value of any storage associated entity. An EQUIVALENCE statement, a COMMON statement, or an ENTRY statement can cause storage association of storage sequences.

An EQUIVALENCE statement causes storage association of data objects only within one scoping unit, unless one of the equivalenced entities is also in a common block (8.10.1.2, 8.10.2.2).

COMMON statements cause data objects in one scoping unit to become storage associated with data objects in another scoping unit.

A common block is permitted to contain a sequence of differing storage units. All scoping units that access named common blocks with the same name shall specify an identical sequence of storage units. Blank common blocks may be declared with differing sizes in different scoping units. For any two blank common blocks, the initial sequence of storage units of the longer blank common block shall be identical to the sequence of storage units of the shorter common block. If two blank common blocks are the same length, they shall have the same sequence of storage units.

An ENTRY statement in a function subprogram causes storage association of the function results that are variables.

Partial association shall exist only between

- an object that is default character or of character sequence type and an object that is default character or...
of character sequence type, or

- an object that is default complex, double precision real, or of numeric sequence type and an object that is
default integer, default real, default logical, double precision real, default complex, or of numeric sequence
type.

For noncharacter entities, partial association may occur only through the use of COMMON, EQUIVALENCE, or ENTRY statements.
For character entities, partial association may occur only through argument association or the use of COMMON or
EQUIVALENCE statements.

Partial association of character entities occurs when some, but not all, of the storage units of the entities are the
same.

A storage unit shall not be explicitly initialized more than once in a program. Explicit initialization overrides
default initialization, and default initialization for an object of derived type overrides default initialization for
a component of the object (7.5.4.6). Default initialization may be specified for a storage unit that is storage
associated provided the objects supplying the default initialization are of the same type and type parameters,
and supply the same value for the storage unit.

19.5.4 Inheritance association

Inheritance association occurs between components of the parent component and components inherited by type
extension into an extended type (7.5.7.2). This association is persistent; it is not affected by the accessibility of the
inherited components.

19.5.5 Establishing associations

When an association is established between two entities by argument association, host association, or construct
association, certain properties of the associating entity become those of the pre-existing entity.

For argument association, the pre-existing entity is the effective argument and the associating entity is the dummy
argument.

For host association, the associating entity is the entity in the contained scoping unit. When a procedure is
invoked, the pre-existing entity that participates in the association is the one from its host instance (15.6.2.4).
Otherwise the pre-existing entity that participates in the association is the entity in the host scoping unit.

For construct association, the associating entity is identified by the associate name and the pre-existing entity is
the selector.

When an association is established by argument association, host association, or construct association, the fol-
lowing applies.

- If the entities have the POINTER attribute, the pointer association status of the associating entity becomes
  the same as that of the pre-existing entity. If the pre-existing entity has a pointer association status of
  associated, the associating entity becomes pointer associated with the same target and, if they are arrays,
  the bounds of the associating entity become the same as those of the pre-existing entity.

- If the associating entity has the ALLOCATABLE attribute, its allocation status becomes the same as that
  of the pre-existing entity. If the pre-existing entity is allocated, the bounds (if it is an array), values of
defferred type parameters, definition status, and value (if it is defined) become the same as those of the
pre-existing entity. If the associating entity is polymorphic and the pre-existing entity is allocated, the
dynamic type of the associating entity becomes the same as that of the pre-existing entity.

- If the associating entity is neither a pointer nor allocatable, its definition status, value (if it is defined), and
dynamic type (if it is polymorphic) become the same as those of the pre-existing entity. If the entities are
arrays and the association is not argument association, the bounds of the associating entity become the
same as those of the pre-existing entity.
- If the associating entity is a pointer dummy argument and the pre-existing entity is a nonpointer actual
argument the associating entity becomes pointer associated with the pre-existing entity and, if the entities
are arrays, the bounds of the associating entity become the same as those of the pre-existing entity.

19.6 Definition and undefinition of variables

19.6.1 Definition of objects and subobjects

A variable may be defined or may be undefined and its definition status may change during execution of a
program. An action that causes a variable to become undefined does not imply that the variable was previously
defined. An action that causes a variable to become defined does not imply that the variable was previously
undefined.

Arrays, including sections, and variables of derived, character, or complex type are objects that consist of zero
or more subobjects. Associations may be established between variables and subobjects and between subobjects
of different variables. These subobjects may become defined or undefined.

An array is defined if and only if all of its elements are defined.

A derived-type scalar object is defined if and only if all of its nonpointer components are defined.

A complex or character scalar object is defined if and only if all of its subobjects are defined.

If an object is undefined, at least one (but not necessarily all) of its subobjects are undefined.

19.6.2 Variables that are always defined

Zero-sized arrays and zero-length strings are always defined.

19.6.3 Variables that are initially defined

The following variables are initially defined:

(1) variables specified to have initial values by DATA statements;
(2) variables specified to have initial values by type declaration statements;
(3) nonpointer default-initialized subcomponents of saved variables that do not have the ALLOCAT-
ABLE or POINTER attribute;
(4) pointers specified to be initially associated with a variable that is initially defined;
(5) variables that are always defined;
(6) variables with the BIND attribute that are initialized by means other than Fortran.
NOTE 19.11

Fortran code:

module mod
  integer, bind(c,name="blivet") :: foo
end module mod

C code:

int blivet = 123;

In the above example, the Fortran variable foo is initially defined to have the value 123 by means other than Fortran.

19.6.4 Variables that are initially undefined

Variables that are not initially defined are initially undefined.

19.6.5 Events that cause variables to become defined

Variables become defined by the following events.

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

2. Execution of a masked array assignment or FORALL assignment statement might cause some or all of the array elements in the assignment statement to become defined (10.2.3).

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. (See (4) in 19.6.6.) Execution of a WRITE statement whose unit specifier identifies an internal file causes each record that is written to become defined.

4. Execution of a DO statement causes the DO variable, if any, to become defined.

5. Beginning of execution of the action specified by an io-implied-do in a synchronous data transfer statement causes the do-variable to become defined.

6. A reference to a procedure causes an entire dummy data object to become defined if the dummy data object does not have INTENT (OUT) and the entire effective argument is defined.

   A reference to a procedure causes a subobject of a dummy argument to become defined if the dummy argument does not have INTENT (OUT) and the corresponding subobject of the effective argument is defined.

7. Execution of an input/output statement containing an IOSTAT= specifier causes the specified integer variable to become defined.

8. Execution of a synchronous input statement containing a SIZE= specifier causes the specified integer variable to become defined.

9. Execution of a wait operation (12.7.1) corresponding to an asynchronous input statement containing a SIZE= specifier causes the specified integer variable to become defined.

10. 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.
(11) If an error, end-of-file, or end-of-record condition occurs during execution of an input/output statement that has an IOMSG= specifier, the iomsg-variable becomes defined.

(12) 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. When an entity of double precision real type becomes defined, all totally associated entities of double precision real type become defined.

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

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

(14) 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.

(15) When all components of a structure of a numeric sequence type or character sequence type become defined as a result of partially associated objects becoming defined, the structure becomes defined.

(16) Execution of a statement with a STAT= specifier causes the variable specified by the STAT= specifier to become defined.

(17) If an error condition occurs during execution of a statement that has an ERRMSG= specifier, the variable specified by the ERRMSG= specifier becomes defined.

(18) Allocation of a zero-sized array or zero-length character variable causes the array or variable to become defined.

(19) Allocation of an object that has a nonpointer default-initialized subcomponent, except by an ALLOCATE statement with a SOURCE= specifier, causes that subcomponent to become defined.

(20) Successful execution of an ALLOCATE statement with a SOURCE= specifier causes a subobject of the allocated object to become defined if the corresponding subobject of the SOURCE= expression is defined.

(21) Invocation of a procedure causes any automatic data object of zero size or zero character length in that procedure to become defined.

(22) When a pointer becomes associated with a target that is defined, the pointer becomes defined.

(23) Invocation of a procedure that contains an unsaved nonpointer nonallocatable local variable causes all nonpointer default-initialized subcomponents of the object to become defined.

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

(25) In a DO CONCURRENT or FORALL construct, the index-name becomes defined when the index-name value set is evaluated.

(26) In a DO CONCURRENT construct, a variable with LOCAL_INIT locality becomes defined at the beginning of each iteration.

(27) An object with the VOLATILE attribute that is changed by a means not specified by the program might become defined (see 8.5.19).

(28) Execution of the BLOCK statement of a BLOCK construct that has an unsaved nonpointer nonallocatable local variable causes all nonpointer default-initialized subcomponents of the variable to become defined.

(29) Execution of an OPEN statement containing a NEWUNIT= specifier causes the specified integer variable to become defined.
Execution of a LOCK statement containing an ACQUIRED_LOCK= specifier causes the specified logical variable to become defined. If the logical variable becomes defined with the value true, the lock variable in the LOCK statement also becomes defined.

Successful execution of a LOCK statement that does not contain an ACQUIRED_LOCK= specifier causes the lock variable to become defined.

Successful execution of an UNLOCK statement causes the lock variable to become defined.

Failure of an image that locked a lock variable without unlocking it causes the lock variable to become defined.

Successful execution of an EVENT POST or EVENT WAIT statement causes the event variable to become defined.

19.6.6 Events that cause variables to become undefined

Variables become undefined by the following events.

1. When a scalar variable of intrinsic type becomes defined, all totally associated variables of different type become undefined. When a double precision scalar variable becomes defined, all partially associated scalar variables become undefined. When a scalar variable becomes undefined, all partially associated double precision scalar variables become undefined.

2. If the evaluation of a function would cause a variable 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 variable becomes undefined when the expression is evaluated.

3. When execution of an instance of a subprogram completes,
   (a) its unsaved local variables become undefined,
   (b) unsaved variables in a named common block that appears in the subprogram become undefined if they have been defined or redefined, unless another active scoping unit is referencing the common block, and
   (c) a variable of type C_PTR from the intrinsic module ISO_C_BINDING whose value is the C address of an unsaved local variable of the subprogram becomes undefined.

4. 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.

5. When an error condition occurs during execution of an output statement in which the unit is an internal file, the internal file becomes 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 io-implied-do, all of the do-variables in the statement become undefined (12.11).

7. 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.

8. Execution of an INQUIRE statement might cause the NAME=, RECL=, and NEXTREC= variables to become undefined (12.10).

9. When a character storage unit becomes undefined, all associated character storage units become undefined.

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

When an unspecified storage unit becomes undefined, all associated unspecified storage units become undefined.

(10) When an allocatable entity is deallocated, it becomes undefined.

(11) When the allocation transfer procedure (16.9.137) causes the allocation status of an allocatable entity to become unallocated, the entity becomes undefined.

(12) Successful execution of an ALLOCATE statement with no SOURCE= specifier causes a subcomponent of an allocated object to become undefined if default initialization has not been specified for that subcomponent.

(13) Successful execution of an ALLOCATE statement with a SOURCE= specifier causes a subobject of the allocated object to become undefined if the corresponding subobject of the SOURCE= expression is undefined.

(14) Execution of an INQUIRE statement causes all inquiry specifier variables to become undefined if an error condition exists, except for any variable in an IOSTAT= or IOMSG= specifier.

(15) When a procedure is invoked
   (a) an optional dummy argument that has no corresponding actual argument becomes undefined,
   (b) a dummy argument with INTENT (OUT) becomes undefined except for any nonpointer default-initialized subcomponents of the argument,
   (c) an actual argument corresponding to a dummy argument with INTENT (OUT) becomes undefined except for any nonpointer default-initialized subcomponents of the argument,
   (d) a subobject of a dummy argument that does not have INTENT (OUT) becomes undefined if the corresponding subobject of the effective argument is undefined, and
   (e) a variable that is the function result of that procedure becomes undefined except for any of its nonpointer default-initialized subcomponents.

(16) When the association status of a pointer becomes undefined or disassociated (19.5.2.4, 19.5.2.5), the pointer becomes undefined.

(17) When a DO CONCURRENT construct terminates, a variable that is defined or becomes undefined during more than one iteration of the construct becomes undefined.

(18) When execution of an iteration of a DO CONCURRENT construct completes, a construct entity of that construct which has LOCAL or LOCAL_INIT locality becomes undefined.

(19) Execution of an asynchronous READ statement causes all of the variables specified by the input list or SIZE= specifier to become undefined. Execution of an asynchronous namelist READ statement causes any variable in the namelist group to become undefined if that variable will subsequently be defined during the execution of the READ statement or the corresponding wait operation (12.7.1).

(20) When a variable with the TARGET attribute is deallocated, a variable of type C_PTR from the intrinsic module ISO_C_BINDING becomes undefined if its value is the C address of any part of the variable that is deallocated.

(21) When a pointer is deallocated, a variable of type C_PTR from the intrinsic module ISO_C_BINDING becomes undefined if its value is the C address of any part of the target that is deallocated.

(22) Execution of the allocation transfer procedure (16.9.137) where the argument TO does not have the TARGET attribute causes a variable of type C_PTR from the intrinsic module ISO_C_BINDING to become undefined if its value is the C address of any part of the argument FROM.
(23) When a BLOCK construct completes execution,
   • its unsaved local variables become undefined, and
   • a variable of type C_PTR from the intrinsic module ISO_C_BINDING, whose value is the C
     address of an unsaved local variable of the BLOCK construct, becomes undefined.

(24) When execution of the host instance of the target of a variable of type C_FUNPTR from the intrinsic
    module ISO_C_BINDING is completed by execution of a RETURN or END statement, the variable
    becomes undefined.

(25) Execution of an intrinsic assignment of the type C_PTR or C_FUNPTR from the intrinsic module
    ISO_C_BINDING in which the variable and expr are not on the same image causes the variable to
    become undefined.

(26) An object with the VOLATILE attribute (8.5.19) might become undefined by means not specified by
    the program.

(27) When a pointer becomes associated with a target that is undefined, the pointer becomes undefined.

(28) When an image fails during execution of a segment, a data object on a nonfailed image becomes
    undefined if it is not a lock variable and it might become undefined by execution of a statement of
    the segment other than an invocation of an atomic subroutine with the object as an actual argument
    corresponding to the ATOM dummy argument.

NOTE 19.12
Execution of a defined assignment statement could leave all or part of the variable undefined.

19.6.7 Variable definition context

Some variables are prohibited from appearing in a syntactic context that would imply definition or undefinition
of the variable (8.5.10, 8.5.15, 15.7). The following are the contexts in which the appearance of a variable implies
such definition or undefinition of the variable:

(1) the variable of an assignment-stmt;

(2) a do-variable in a do-stmt or io-implied-do;

(3) an input-item in a read-stmt;

(4) a variable-name in a namelist-stmt if the namelist-group-name appears in a NML= specifier in a
    read-stmt;

(5) an internal-file-variable in a write-stmt;

(6) an IOSTAT=, SIZE=, or IOMSG= specifier in an input/output statement;

(7) a specifier in an INQUIRE statement other than FILE=, ID=, and UNIT=;

(8) a NEWUNIT= specifier in an OPEN statement;

(9) a stat-variable, allocate-object, or errmsg-variable;

(10) an actual argument in a reference to a procedure with an explicit interface if the corresponding
     dummy argument is not a pointer and has INTENT (OUT) or INTENT (INOUT);

(11) a variable that is a selector in an ASSOCIATE, CHANGE TEAM, SELECT RANK, or SELECT
    TYPE construct if the corresponding associate name or any subobject thereof appears in a variable
    definition context;

(12) an event-variable in an EVENT POST or EVENT WAIT statement;

(13) a lock-variable in a LOCK or UNLOCK statement;
(14) a scalar-logical-variable in an ACQUIRED_LOCK= specifier;

(15) a team-variable in a FORM TEAM statement.

2 If a reference to a function appears in a variable definition context the result of the function reference shall be a pointer that is associated with a definable target. That target is the variable that becomes defined or undefined.

19.6.8 Pointer association context

1 Some pointers are prohibited from appearing in a syntactic context that would imply alteration of the pointer association status (19.5.2.2, 8.5.10, 8.5.15, 15.7). The following are the contexts in which the appearance of a pointer implies such alteration of its pointer association status:

• a pointer-object in a nullify-stmt;

• a data-pointer-object or proc-pointer-object in a pointer-assignment-stmt;

• an allocate-object in an allocate-stmt or deallocate-stmt;

• an actual argument in a reference to a procedure if the corresponding dummy argument is a pointer with the INTENT (OUT) or INTENT (INOUT) attribute.
Annex A

(Informative)

Processor Dependencies

A.1 Unspecified Items

This document does not specify the following:

• the properties excluded in 1;
• a processor’s error detection capabilities beyond those listed in 4.2;
• which additional intrinsic procedures or modules a processor provides (4.2);
• the number and kind of companion processors (5.5.7);
• the number of representation methods and associated kind type parameter values of the intrinsic types (7.4), except that there shall be at least two representation methods for type real, and a representation method of type complex that corresponds to each representation method for type real.

A.2 Processor Dependencies

According to this document, the following are processor dependent:

• the order of evaluation of the specification expressions within the specification part of an invoked Fortran procedure (5.3.5);
• how soon an image terminates if another image initiates error termination (5.3.5);
• the value of a reference to a coindexed object on a failed image (5.3.6);
• the conditions that cause an image to fail (5.3.6);
• whether the processor supports a concept of process exit status, and if so, the process exit status on program termination (5.3.7);
• the mechanism of a companion processor, and the means of selecting between multiple companion processors (5.5.7);
• the processor character set (6.1);
• the means for specifying the source form of a program unit (6.3);
• the maximum number of characters allowed on a source line containing characters not of default kind (6.3.2, 6.3.3);
• the maximum depth of nesting of include lines (6.4);
• the interpretation of the char-literal-constant in the include line (6.4);
• the set of values supported by an intrinsic type, other than logical (7.1.3);
• the kind type parameter value of a complex literal constant, if both the real part and imaginary part are of type real with the same precision, but have different kind type parameter values (7.4.3.3);
• the kind of a character length type parameter (7.4.4.1);
• the blank padding character for nondefault character kind (7.4.4.2).
• whether particular control characters can appear within a character literal constant in fixed source form (7.4.4.3);
• the collating sequence for each character set (7.4.4.4);
• the order of finalization of components of objects of derived type (7.5.6.2);
• the order of finalization when several objects are finalized as the consequence of a single event (7.5.6.2);
• whether and when an object is finalized if it is allocated by pointer allocation and it later becomes unreachable due to all pointers associated with the object having their pointer association status changed (7.5.6.3);
• whether an object is finalized by a deallocation in which an error condition occurs (7.5.6.3);
• the kind type parameter of each enumeration and its enumerators (7.6);
• whether an array is contiguous, except as specified in 8.5.7;
• the set of error conditions that can occur in ALLOCATE and DEALLOCATE statements (9.7.1, 9.7.3);
• the allocation status of a variable after evaluation of an expression if the evaluation of a function would change the allocation status of the variable and if a reference to the function appears in the expression in which the value of the function is not needed to determine the value of the expression (9.7.1.3);
• the order of deallocation when several objects are deallocated by a DEALLOCATE statement (9.7.3);
• the order of deallocation when several objects are deallocated due to the occurrence of an event described in 9.7.3.2;
• whether an allocated allocatable subobject is deallocated when an error condition occurs in the deallocation of an object (9.7.3.2);
• the positive integer values assigned to the stat-variable in a STAT= specifier as the result of an error condition (9.7.4, 11.6.11);
• the allocation status or pointer association status of an allocate-object if an error occurs during execution of an ALLOCATE or DEALLOCATE statement (9.7.4);
• the value assigned to the errmsg-variable in an ERRMSG= specifier as the result of an error condition (9.7.5, 11.6.11);
• the kind type parameter value of the result of a numeric intrinsic binary operation where
  – both operands are of type integer but with different kind type parameters, and the decimal exponent ranges are the same,
  – one operand is of type real or complex and the other is of type real or complex with a different kind type parameter, and the decimal precisions are the same,
and for a logical intrinsic binary operation where the operands have different kind type parameters (10.1.9.3);
• the character assigned to the variable in an intrinsic assignment statement if the kind of the expression is different and the character is not representable in the kind of the variable (10.2.1.3);
• the order of evaluation of the specification expressions within the specification part of a BLOCK construct when the construct is executed (11.1.4);
• the pointer association status of a pointer that has its pointer association changed in more than one iteration of a DO CONCURRENT construct, on termination of the construct (11.1.7);
• the ordering between records written by different iterations of a DO CONCURRENT construct if the records are written to a file connected for sequential access by more than one iteration (11.1.7);
• the manner in which the stop code of a STOP or ERROR STOP statement is made available (11.4);
• the mechanisms available for creating dependencies for cooperative synchronization (11.6.5);
• the value of the count of the event variable in an EVENT POST or EVENT WAIT statement if an error
condition occurs (11.6.7, 11.6.8);
• the image index value established for each image in a team by a FORM TEAM statement without a
NEW_INDEX= specifier (11.6.9);
• the set of error conditions that can occur in image control statements (11.6.11);
• the relationship between the file storage units when viewing a file as a stream file, and the records when
viewing that file as a record file (12);
• whether particular control characters can appear in a formatted record or a formatted stream file (12.2.2);
• the form of values in an unformatted record (12.2.3);
• at any time, the set of allowed access methods, set of allowed forms, set of allowed actions, and set of
allowed record lengths for a file (12.3);
• the set of allowable names for a file (12.3);
• whether a named file on one image is the same as a file with the same name on another image (12.3.1);
• the set of external files that exist for a program (12.3.2);
• the relationship between positions of successive file storage units in an external file that is connected for
formatted stream access (12.3.3.4);
• the external unit preconnected for sequential formatted input and identified by an asterisk or the named
constant INPUT_UNIT of the ISO_FORTRAN_ENV intrinsic module (12.5);
• the external unit preconnected for sequential formatted output and identified by an asterisk or the named
constant OUTPUT_UNIT of the ISO_FORTRAN_ENV intrinsic module (12.5);
• the external unit preconnected for sequential formatted output and identified by the named constant ER-
ROR_UNIT of the ISO_FORTRAN_ENV intrinsic module, and whether this unit is the same as OUT-
PUT_UNIT (12.5);
• at any time, the set of external units that exist for an image (12.5.3);
• whether a unit can be connected to a file that is also connected to a C stream (12.5.4);
• whether a file can be connected to more than one unit at the same time (12.5.4);
• the effect of performing input/output operations on multiple units while they are connected to the same
external file (12.5.4);
• the result of performing input/output operations on a unit connected to a file that is also connected to a C
stream (12.5.4);
• whether the files connected to the units INPUT_UNIT, OUTPUT_UNIT, and ERROR_UNIT correspond
to the predefined C text streams standard input, standard output, and standard error, respectively (12.5.4);
• the results of performing input/output operations on an external file both from Fortran and from a procedure
defined by means other than Fortran (12.5.4);
• the default value for the ACTION= specifier in an OPEN statement (12.5.6.4);
• the encoding of a file opened with ENCODING='DEFAULT' (12.5.6.9);
• the file connected by an OPEN statement with STATUS='SCRATCH' (12.5.6.10);
• the interpretation of case in a file name (12.5.6.10, 12.10.2.2);
• the position of a file after executing an OPEN statement with a POSITION= specifier of ASIS, when the
file previously existed but was not connected (12.5.6.14);
• the default value for the RECL= specifier in an OPEN statement (12.5.6.15);
• the effect of RECL= on a record containing any nondefault characters (12.5.6.15);
• the default input/output rounding mode (12.5.6.16);
• the default sign mode (12.5.6.17);
• the file status when STATUS='UNKNOWN' is specified in an OPEN statement (12.5.6.18);
• the value assigned to the variable in the ID= specifier in an asynchronous data transfer statement when execution of the statement is successfully completed (12.6.2.9);
• whether POS= is permitted with particular files, and whether POS= can position a particular file to a position prior to its current position (12.6.2.11);
• the form in which a single value of derived type is treated in an unformatted input/output statement if the effective item is not processed by a defined input/output procedure (12.6.3);
• the result of unformatted input when the value stored in the file has a different type or type parameters from that of the input list item (12.6.4.5.2);
• the negative value of the unit argument to a defined input/output procedure if the parent data transfer statement accesses an internal file (12.6.4.8.2);
• the manner in which the processor makes the value of the iomsg argument of a defined input/output procedure available if the procedure assigns a nonzero value to the iostat argument and the processor therefore terminates execution of the program (12.6.4.8.2);
• the action caused by the flush operation, whether the processor supports the flush operation for the specified unit, and the negative value assigned to the IOSTAT= variable if the processor does not support the flush operation for the specified unit (12.9);
• the case of characters assigned to the variable in a NAME= specifier in an INQUIRE statement (12.10.2.15);
• which of the connected external unit numbers is assigned to the scalar-int-variable in the NUMBER= specifier in an INQUIRE by file statement, if more than one unit on an image is connected to the file (12.10.2.18);
• the value of the variable in a POSITION= specifier in an INQUIRE statement if the file has been repositioned since connection (12.10.2.23);
• the relationship between file size and the data stored in records in a sequential or direct access file (12.10.2.30);
• the number of file storage units needed to store data in an unformatted file (12.10.3);
• the set of error conditions that can occur in input/output statements (12.11.1);
• when an input/output error condition occurs or is detected (12.11.1);
• the positive integer value assigned to the variable in an IOSTAT= specifier as the result of an error condition (12.11.5);
• the value assigned to the variable in an IOMSG= specifier as the result of an error condition (12.11.6);
• the result of output of non-representable characters to a Unicode file (13.7.1);
• the interpretation of the optional non-blank characters within the parentheses of a real NaN input field (13.7.2.3.2);
• the interpretation of a sign in a NaN input field (13.7.2.3.2);
• for output of an IEEE NaN, whether after the letters 'NaN', the processor produces additional alphanumeric characters enclosed in parentheses (13.7.2.3.2);
• the effect of the input/output rounding mode PROCESSOR_DEFINED (13.7.2.3.8);
• which value is chosen if the input/output rounding mode is NEAREST and the value to be converted is exactly halfway between the two nearest representable values in the result format (13.7.2.3.8);
• the field width, decimal part width, and exponent width used for the G0 edit descriptor (13.7.5);
• the file position when position editing skips a character of nondefault kind in an internal file of default character kind or an external unit that is not connected to a Unicode file (13.8.1.1);
• when the sign mode is PROCESSOR_DEFINED, whether a plus sign appears in a numeric output field for a nonnegative value (13.8.4);
• the results of list-directed output (13.10.4);
• the results of namelist output (13.11.4);
• the interaction between argument association and pointer association, (15.5.2.4);
• the values returned by some intrinsic functions (16);
• how the sequences of atomic actions in unordered segments interleave (16.5);
• the value assigned to a STAT argument in a reference to an atomic subroutine when an error condition occurs (16.5);
• the effects of calling COMMAND_ARGUMENT_COUNT, EXECUTE_COMMAND_LINE, GET_COMMAND, and GET_COMMAND_ARGUMENT on any image other than image 1 in the initial team (16.7);
• whether each image uses a separate random number generator, or if some or all images use common random number generators (16.7);
• whether the results returned from CPU_TIME, DATE_AND_TIME and SYSTEM_CLOCK are dependent on which image calls them (16.7);
• the set of error conditions that can occur in some intrinsic subroutines (16.9);
• the value assigned to a CMDSTAT, ERRMSG, EXITSTAT, STAT, or STATUS argument to indicate a processor-dependent error condition (16.9);
• the computed value of the the intrinsic subroutine CO_REDUCE (16.9.49) and the intrinsic subroutine CO_SUM (16.9.50);
• the value assigned to the TIME argument by the intrinsic subroutine CPU_TIME (16.9.57);
• whether date, clock, and time zone information is available (16.9.59);
• whether date, clock, and time zone information on one image is the same as that on another image (16.9.59);
• the value of command argument zero, if the processor does not support the concept of a command name (16.9.83);
• the order of command arguments (16.9.83);
• whether the significant length of a command argument includes trailing blanks (16.9.83);
• the interpretation of case for the NAME argument of the intrinsic subroutine GET_ENVIRONMENT_VARIABLE (16.9.84);
• whether an environment variable that exists on an image also exists on another image, and if it does exist on both images, whether the values are the same or different (16.9.84);
• the value assigned to the pseudorandom number generator by the intrinsic subroutine RANDOM_INIT (16.9.155);
• the computation of the seed value used by the pseudorandom number generator (16.9.157);
• on images that use a common random number generator, the interleaving of values assigned by RANDOM_NUMBER in unordered segments (16.7);
• the value assigned to the seed by the intrinsic subroutine RANDOM_SEED when no argument is present (16.9.157);
• the values assigned to its arguments by the intrinsic subroutine SYSTEM_CLOCK (16.9.186);
• the values of the named constants in the intrinsic module ISO_FORTRAN_ENV (16.10.2);
• the values returned by the functions COMPILER_OPTIONS and COMPILER_VERSION in the intrinsic
  module ISO_FORTRAN_ENV (16.10.2);
• the extent to which a processor supports IEEE arithmetic (17);
• whether a flag that is quiet on entry to a scoping unit that does not access IEEE_FEATURES, IEEE_-
  EXCEPTIONS, or IEEE_ARITHMETIC is signaling on exit (17.1);
• the conditions under which IEEE_OVERFLOW is raised in a calculation involving non-ISO/IEC/IEEE
  60559:2011 floating-point data (17.3);
• the conditions under which IEEE_OVERFLOW and IEEE_DIVIDE_BY_ZERO are raised in a floating-
  point exponentiation operation (17.3);
• the conditions under which IEEE_DIVIDE_BY_ZERO is raised in a calculation involving non-ISO/IEC/IEEE
  60559:2011 floating-point data (17.3);
• whether an exception signals at the end of a sequence of statements that has no invocations of IEEE_GET_-_FLAG, IEEE_SET_FLAG, IEEE_GET_STATUS, IEEE_SET_STATUS, or IEEE_SET_HALTING_-MODE, in which execution of an operation would cause it to signal, if no value of a variable depends upon
  the result of the operation (17.3);
• the initial rounding modes (17.4);
• the initial underflow mode (17.5);
• the initial halting mode (17.6);
• whether IEEE_INT implements the convertToInteger{round} or convertToIntegerExact{round} operation
  specified by ISO/IEC 60559:2011 (17.11.11);
• which argument is the result value of IEEE_MAX_NUM, IEEE_MAX_NUM_MAG, IEEE_MIN_NUM,
  or IEEE_MIN_NUM_MAG when both arguments are quiet NaNs (17.11.17, 17.11.18, 17.11.19, 17.11.20);
• the requirements on the storage sequence to be associated with the pointer FPTR by the C_F_POINTER
  subroutine (18.2.3.4);
• the order of the members of the CFI_dim_t structure defined in the source file CFI_Fortran_binding.h
  (18.5.2);
• members of the CFI_cdesc_t structure defined in the source file CFI_Fortran_binding.h beyond the re-
  quirements of 18.5.3;
• the value of CFI_MAX_RANK in the source file CFI_Fortran_binding.h (18.5.4);
• the value of CFI_VERSION in the source file CFI_Fortran_binding.h (18.5.4);
• which error condition is detected if more than one error condition could be detected for an invocation of
  one of the functions declared in the source file CFI_Fortran_binding.h (18.5.5.1);
• the values of the attribute specifier macros defined in the source file CFI_Fortran_binding.h (18.5.4);
• the values of the type specifier macros defined in the source file CFI_Fortran Binding.h;
• which additional type specifier values are defined in the source file CFI_Fortran_binding.h (18.5.4);
• the values of the error code macros other than CFI_SUCCESS that are defined in the source file CFI_-
  Fortran_binding.h (18.5.4);
• the base address of a zero-sized array (18.5.3);
• the values of the floating-point exception flags on entry to a procedure defined by means other than Fortran
  (18.10.3);
• whether a procedure defined by means other than Fortran is an asynchronous communication initiation or
  completion procedure (18.10.4).
Annex B

(Informative)

Deleted and obsolescent features

B.1 Deleted features from Fortran 90

1 These deleted features are those features of Fortran 90 that were redundant and considered largely unused.

2 The following Fortran 90 features are not required.

(1) Real and double precision DO variables.

In Fortran 77 and Fortran 90, a DO variable was allowed to be of type real or double precision in addition to type integer; this has been deleted. A similar result can be achieved by using a DO construct with no loop control and the appropriate exit test.

(2) Branching to an END IF statement from outside its block.

In Fortran 77 and Fortran 90, it was possible to branch to an END IF statement from outside the IF construct; this has been deleted. A similar result can be achieved by branching to a CONTINUE statement that is immediately after the END IF statement.

(3) PAUSE statement.

The PAUSE statement, provided in Fortran 66, Fortran 77, and Fortran 90, has been deleted. A similar result can be achieved by writing a message to the appropriate unit, followed by reading from the appropriate unit.

(4) ASSIGN and assigned GO TO statements, and assigned format specifiers.

The ASSIGN statement and the related assigned GO TO statement, provided in Fortran 66, Fortran 77, and Fortran 90, have been deleted. Further, the ability to use an assigned integer as a format, provided in Fortran 77 and Fortran 90, has been deleted. A similar result can be achieved by using other control constructs instead of the assigned GO TO statement and by using a default character variable to hold a format specification instead of using an assigned integer.

(5) H edit descriptor.

In Fortran 77 and Fortran 90, there was an alternative form of character string edit descriptor, which had been the only such form in Fortran 66; this has been deleted. A similar result can be achieved by using a character string edit descriptor.

(6) Vertical format control.

In Fortran 66, Fortran 77, Fortran 90, and Fortran 95 formatted output to certain units resulted in the first character of each record being interpreted as controlling vertical spacing. There was no standard way to detect whether output to a unit resulted in this vertical format control, and no way to specify that it should be applied; this has been deleted. The effect can be achieved by post-processing a formatted file.

3 See ISO/IEC 1539:1991 for detailed rules of how these deleted features worked.
B.2 Deleted features from Fortran 2008

These deleted features are those features of Fortran 2008 that were redundant and considered largely unused.

The following Fortran 2008 features are not required.

1. Arithmetic IF statement.
   The arithmetic IF statement is incompatible with ISO/IEC/IEEE 60559:2011 and necessarily involves
   the use of statement labels; statement labels can hinder optimization, and make code hard to read
   and maintain. Similar logic can be more clearly encoded using other conditional statements.

2. Nonblock DO construct
   The nonblock forms of the DO loop were confusing and hard to maintain. Shared termination and
   dual use of labeled action statements as do termination and branch targets were especially error-
   prone.

B.3 Obsolescent features

B.3.1 General

The obsolescent features are those features of Fortran 90 that were redundant and for which better methods were
available in Fortran 90. Subclause 4.4.3 describes the nature of the obsolescent features. The obsolescent features
in this document are the following.

1. Alternate return — see B.3.2.
2. Computed GO TO — see B.3.3.
3. Statement functions — see B.3.4.
4. DATA statements amongst executable statements — see B.3.5.
5. Assumed length character functions — see B.3.6.
6. Fixed form source — see B.3.7.
7. CHARACTER* form of CHARACTER declaration — see B.3.8.
8. ENTRY statements — see B.3.9.
10. COMMON and EQUIVALENCE statements, and the block data program unit — see B.3.11.
11. Specific names for intrinsic functions — see B.3.12.
12. FORALL construct and statement — see B.3.13

B.3.2 Alternate return

An alternate return introduces labels into an argument list to allow the called procedure to direct the execution
of the caller upon return. The same effect can be achieved with a return code that is used in a SELECT CASE
construct on return. This avoids an irregularity in the syntax and semantics of argument association. For example,

\[
\text{CALL SUBR_NAME (X, Y, Z, *100, *200, *300)}
\]

can be replaced by

\[
\text{CALL SUBR_NAME (X, Y, Z, RETURN_CODE)} \quad \text{SELECT CASE (RETURN_CODE)}
\]
CASE (1)
... 
CASE (2)
...
CASE (3)
...
CASE DEFAULT
...
END SELECT

B.3.3 Computed GO TO statement

The computed GO TO statement has been superseded by the SELECT CASE construct, which is a generalized, easier to use, and clearer means of expressing the same computation.

B.3.4 Statement functions

Statement functions are subject to a number of nonintuitive restrictions and are a potential source of error because their syntax is easily confused with that of an assignment statement.

B.3.5 DATA statements among executables

The statement ordering rules allow DATA statements to appear anywhere in a program unit after the specification statements. The ability to position DATA statements amongst executable statements is very rarely used, unnecessary, and a potential source of error.

B.3.6 Assumed character length functions

Assumed character length for functions is an irregularity in the language in that elsewhere in Fortran the philosophy is that the attributes of a function result depend only on the actual arguments of the invocation and on any data accessible by the function through host or use association. Some uses of this facility can be replaced with an automatic character length function, where the length of the function result is declared in a specification expression. Other uses can be replaced by the use of a subroutine whose arguments correspond to the function result and the function arguments.

B.3.7 Fixed form source

Fixed form source was designed when the principal machine-readable input medium for new programs was punched cards. Now that new and amended programs are generally entered via keyboards with screen displays, it is an unnecessary overhead, and is potentially error-prone, to have to locate positions 6, 7, or 72 on a line. Free form source was designed expressly for this more modern technology.

It is a simple matter for a software tool to convert from fixed to free form source.
B.3.8 CHARACTER* form of CHARACTER declaration

In addition to the CHARACTER*\textit{char-length} form introduced in FORTRAN 77, Fortran 90 provided the CHARACTER([ LEN = | type-param-value]) form. The older form (CHARACTER*\textit{char-length}) is redundant.

B.3.9 ENTRY statements

ENTRY statements allow more than one entry point to a subprogram, facilitating sharing of data items and executable statements local to that subprogram.

This can be replaced by a module containing the (private) data items, with a module procedure for each entry point and the shared code in a private module procedure.

B.3.10 Label DO statement

The label in the DO statement is redundant with the construct name. Furthermore, the label allows unrestricted branches and, for its main purpose (the target of a conditional branch to skip the rest of the current iteration), is redundant with the CYCLE statement, which is clearer.

B.3.11 COMMON and EQUIVALENCE statements and the block data program unit

Common blocks are error-prone and have largely been superseded by modules. EQUIVALENCE similarly is error-prone. Whilst use of these statements was invaluable prior to Fortran 90 they are now redundant and can inhibit performance. The block data program unit exists only to serve common blocks and hence is also redundant.

B.3.12 Specific names for intrinsic functions

The specific names of the intrinsic functions are often obscure and hinder portability. They have been redundant since Fortran 90. Use generic names for references to intrinsic procedures.

B.3.13 FORALL construct and statement

The FORALL construct and statement were added to the language in the expectation that they would enable highly efficient execution, especially on parallel processors. However, experience indicates that they are too complex and have too many restrictions for compilers to take advantage of them. They are redundant with the DO CONCURRENT construct, and many of the manipulations for which they might be used can be done more effectively using pointers, especially using pointer rank remapping.
Annex C

(Informative)

Extended notes

C.1 Fortran 2008 features not mentioned in its Introduction

1 The following features were new in Fortran 2008 but not originally listed in its Introduction as being new features:

- An array or object with a nonconstant length type parameter can have the VALUE attribute.
- Multiple allocations are permitted in a single ALLOCATE statement with the SOURCE= specifier.
- A PROCEDURE statement can have a double colon before the first procedure name.
- An argument to a pure procedure can have default INTENT if it has the VALUE attribute.
- The PROTECTED attribute can be specified by the procedure declaration statement.
- A defined-operator can be used in a specification expression.
- All transformational functions from the intrinsic modules IEEE_ARITHMETIC and IEEE_EXCEPTIONS can be used in constant expressions. All transformational functions from the intrinsic modules IEEE_ARITHMETIC, IEEE_EXCEPTIONS, and ISO_C_BINDING can be used in specification expressions.
- A contiguous array variable that is not interoperable but which has interoperable type and kind type parameter (if any), and a scalar character variable with length greater than 1 and kind C_CHAR in the intrinsic module ISO_C_BINDING, can be used as the argument of the function C_LOC in the intrinsic module ISO_C_BINDING, provided the variable has the POINTER or TARGET attribute.
- The name of an external procedure that has a binding label is a local identifier and not a global identifier.
- A procedure that is not a procedure pointer can be an actual argument that corresponds to a procedure pointer dummy argument with the INTENT (IN) attribute.

2 All but the last of the above list were subsequently added to the Introduction by Technical Corrigenda.

C.2 Clause 7 notes

C.2.1 Selection of the approximation methods (7.4.3.2)

1 One can select the real approximation method for an entire program through the use of a module and the parameterized real type. This is accomplished by defining a named integer constant to have a particular kind type parameter value and using that named constant in all real, complex, and derived-type declarations. For example, the specification statements

```fortran
INTEGER, PARAMETER :: LONG_FLOAT = 8
REAL (LONG_FLOAT) X, Y
COMPLEX (LONG_FLOAT) Z
```

specify that the approximation method corresponding to a kind type parameter value of 8 is supplied for the data objects X, Y, and Z in the program unit. The kind type parameter value LONG_FLOAT can be made available...
to an entire program by placing the INTEGER specification statement in a module and accessing the named constant LONG_FLOAT with a USE statement. Note that by changing 8 to 4 once in the module, a different approximation method is selected.

To avoid the use of the processor-dependent values 4 or 8, replace 8 by KIND (0.0) or KIND (0.0D0). Another way to avoid these processor-dependent values is to select the kind value using the intrinsic function SELECTED_REAL_KIND (16.9.170). In the above specification statement, the 8 might be replaced by, for instance, SELECTED_REAL_KIND (10, 50), which requires an approximation method to be selected with at least 10 decimal digits of precision and a range from $10^{-50}$ to $10^{50}$. There are no magnitude or ordering constraints placed on kind values, in order that implementers have flexibility in assigning such values and can add new kinds without changing previously assigned kind values.

As kind values have no portable meaning, a good practice is to use them in programs only through named constants as described above (for example, SINGLE, IEEE_SINGLE, DOUBLE, and QUAD), rather than using the kind values directly.

### C.2.2 Type extension and component accessibility (7.5.2.2, 7.5.4)

The default accessibility of the components of an extended type can be specified in the type definition. The accessibility of its components can be specified individually. For example:

```fortran
module types
  type base_type
    private       !-- Sets default accessibility
    integer :: i  !-- a private component
    integer, private :: j !-- another private component
    integer, public :: k  !-- a public component
  end type base_type

  type, extends(base_type) :: my_type
    private       !-- Sets default for components declared in my_type
    integer :: l  !-- A private component.
    integer, public :: m  !-- A public component.
  end type my_type

end module types

subroutine sub
  use types
  type (my_type) :: x

  ...

  call another_sub( &
    x%base_type, &  !-- ok because base_type is a public subobject of x
    x%base_type%k, &  !-- ok because x%base_type is ok and has k as a
                     !-- public component.
    x%k, &  !-- ok because it is shorthand for x%base_type%k
```

546 ISO/IEC JTC 1/SC 22/WG5/N2118
x%base_type%i, & !-- Invalid because i is private.
x%i) !-- Invalid because it is shorthand for x%base_type%i
end subroutine sub

C.2.3 Generic type-bound procedures (7.5.5)

Example of a derived type with generic type-bound procedures:

The only difference between this example and the same thing rewritten to use generic interface blocks is that with type-bound procedures,

USE rational_numbers, ONLY: rational

does not block the type-bound procedures; the user still gets access to the defined assignment and extended operations.

MODULE rational_numbers
IMPLICIT NONE
PRIVATE
TYPE,PUBLIC :: rational
PRIVATE
INTEGER n,d
CONTAINS
! ordinary type-bound procedure
PROCEDURE :: real => rat_to_real
! specific type-bound procedures for generic support
PROCEDURE,PRIVATE :: rat_asgn_i, rat_plus_i, rat_plus_rat => rat_plus
PROCEDURE,PRIVATE,PASS(b) :: i_plus_rat
! generic type-bound procedures
GENERIC :: ASSIGNMENT(=) => rat_asgn_i
GENERIC :: OPERATOR(+) => rat_plus_rat, rat_plus_i, i_plus_rat
END TYPE
CONTAINS
ELEMENTAL REAL FUNCTION rat_to_real(this) RESULT(r)
CLASS(rational),INTENT(IN) :: this
r = REAL(this%n)/this%d
END FUNCTION
ELEMENTAL SUBROUTINE rat_asgn_i(a,b)
CLASS(rational),INTENT(OUT) :: a
INTEGER,INTENT(IN) :: b
a%n = b
a%d = 1
END SUBROUTINE
ELEMENTAL TYPE(rational) FUNCTION rat_plus_i(a,b) RESULT(r)
CLASS(rational),INTENT(IN) :: a
INTEGER,INTENT(IN) :: b
r%n = a%n + b*a%d
r%d = a%d

END FUNCTION

ELEMENTAL TYPE(rational) FUNCTION i_plus_rat(a,b) RESULT(r)
   INTEGER, INTENT(IN) :: a
   CLASS(rational), INTENT(IN) :: b
   r%n = b%n + a*b%d
   r%d = b%d
END FUNCTION

ELEMENTAL TYPE(rational) FUNCTION rat_plus(a,b) RESULT(r)
   CLASS(rational), INTENT(IN) :: a,b
   r%n = a%n*b%d + b%n*a%d
   r%d = a%d*b%d
END FUNCTION

END

C.2.4 Abstract types (7.5.7.1)

The following illustrates how an abstract type can be used as the basis for a collection of related types, and how
a non-abstract member of that collection can be created by type extension.

TYPE, ABSTRACT :: DRAWABLE_OBJECT
   REAL, DIMENSION(3) :: RGB_COLOR = (/1.0,1.0,1.0/) ! White
   REAL, DIMENSION(2) :: POSITION = (/0.0,0.0/) ! Centroid
CONTAINS
   PROCEDURE(RENDER_X), PASS(OBJECT), DEFERRED :: RENDER
END TYPE DRAWABLE_OBJECT

ABSTRACT INTERFACE
   SUBROUTINE RENDER_X(OBJECT, WINDOW)
      IMPORT DRAWABLE_OBJECT, X_WINDOW
      CLASS(DRAWABLE_OBJECT), INTENT(IN) :: OBJECT
      CLASS(X_WINDOW), INTENT(INOUT) :: WINDOW
   END SUBROUTINE RENDER_X
END INTERFACE

... 

TYPE, EXTENDS(DRAWABLE_OBJECT) :: DRAWABLE_TRIANGLE ! Not ABSTRACT
   REAL, DIMENSION(2,3) :: VERTICES ! In relation to centroid
CONTAINS
   PROCEDURE, PASS(OBJECT) :: RENDER=>RENDER_TRIANGLE_X
END TYPE DRAWABLE_TRIANGLE

The actual drawing procedure will draw a triangle in WINDOW with vertices at x and y coordinates at
OBJECT%POSITION(1)+OBJECT%VERTICES(1,1:3) and OBJECT%POSITION(2)+OBJECT%VERTICES(2,1:3): 

SUBROUTINE RENDER_TRIANGLE_X(OBJECT, WINDOW)
   CLASS(DRAWABLE_TRIANGLE), INTENT(IN) :: OBJECT
CLASS(X_WINDOW), INTENT(INOUT) :: WINDOW

END SUBROUTINE RENDER_TRIANGLE_X

C.2.5 Structure constructors and generic names (7.5.10)

A generic name can be the same as a type name. This can be used to emulate user-defined structure constructors for that type, even if the type has private components. For example:

MODULE mytype_module
  TYPE mytype
    PRIVATE
    COMPLEX value
    LOGICAL exact
  END TYPE
  INTERFACE mytype
    MODULE PROCEDURE int_to_mytype
  END INTERFACE
  CONTAINS
    TYPE(mytype) FUNCTION int_to_mytype(i)
      INTEGER, INTENT(IN) :: i
      int_to_mytype%value = i
      int_to_mytype%exact = .TRUE.
    END FUNCTION
  END INTERFACE
  ! Operator definitions etc.
  ...
END

PROGRAM example
  USE mytype_module
  TYPE(mytype) x
  x = mytype(17)
END

The type name can still be used as a generic name if the type has type parameters. For example:

MODULE m
  TYPE t(kind)
    INTEGER, KIND :: kind
    COMPLEX(kind) value
  END TYPE
  INTERFACE t
    MODULE PROCEDURE real_to_t1, dble_to_t2, int_to_t1, int_to_t2
  END INTERFACE

ISO/IEC JTC 1/SC 22/WG5/N2118 549
CONTAINS

TYPE(t(single)) FUNCTION real_to_t1(x)
  REAL(single) x
  real_to_t1%value = x
END FUNCTION

TYPE(t(double)) FUNCTION dble_to_t2(x)
  REAL(double) x
  dble_to_t2%value = x
END FUNCTION

TYPE(t(single)) FUNCTION int_to_t1(x,mold)
  INTEGER x
  TYPE(t(single)) mold
  int_to_t1%value = x
END FUNCTION

TYPE(t(double)) FUNCTION int_to_t2(x,mold)
  INTEGER x
  TYPE(t(double)) mold
  int_to_t2%value = x
END FUNCTION

END

PROGRAM example
  USE m
  TYPE(t(single)) x
  TYPE(t(double)) y
  x = t(1.5) ! References real_to_t1
  x = t(17,mold=x) ! References int_to_t1
  y = t(1.5d0) ! References dble_to_t2
  y = t(42,mold=y) ! References int_to_t2
  y = t(kind(0d0)) ((0,1)) ! Uses the structure constructor for type t
END

C.2.6 Final subroutines (7.5.6, 7.5.6.2, 7.5.6.3, 7.5.6.4)

Example of a parameterized derived type with final subroutines:

MODULE m
  TYPE t(k)
    INTEGER, KIND :: k
    REAL(k), POINTER :: vector(:) => NULL()
  CONTAINS
    FINAL :: finalize_t1s, finalize_t1v, finalize_t2e
  END TYPE
  CONTAINS
    SUBROUTINE finalize_t1s(x)
    ...
END
Example of extended types with final subroutines:

```fortran
MODULE m
  TYPE t1
    REAL a, b
  END TYPE
  TYPE, EXTENDS(t1) :: t2
    REAL, POINTER :: c(:), d(:)
  CONTAINS
    FINAL :: t2f
  END TYPE
  TYPE, EXTENDS(t2) :: t3
    REAL, POINTER :: e
  CONTAINS
    FINAL :: t3f
  END TYPE
  CONTAINS
    SUBROUTINE t2f(x) ! Finalizer for TYPE(t2)'s extra components
      ! ...
      ! ...
    END SUBROUTINE
END MODULE
```

```fortran
SUBROUTINE finalize_t1v(x)
  TYPE(t(KIND(0.0))) x(:)
  DO i = LBOUND(x,1), UBOUND(x,1)
    IF (ASSOCIATED(x(i)%vector)) DEALLOCATE(x(i)%vector)
  END DO
END SUBROUTINE

ELEMENTAL SUBROUTINE finalize_t2e(x)
  TYPE(t(KIND(0.0d0))), INTENT(INOUT) :: x
  IF (ASSOCIATED(x%vector)) DEALLOCATE(x%vector)
END SUBROUTINE

SUBROUTINE example(n)
  USE m
  TYPE(t(KIND(0.0))) a, b(10), c(n,2)
  TYPE(t(KIND(0.0d0))) d(n,n)
  ! Returning from this subroutine will effectively do
  CALL finalize_t1s(a)
  CALL finalize_t1v(b)
  CALL finalize_t2e(d)
  ! No final subroutine will be called for variable C because the user
  ! omitted to define a suitable specific procedure for it.
END SUBROUTINE
```
SUBROUTINE example
USE m
TYPE(t1) x1
TYPE(t2) x2
TYPE(t3) x3
...
! Returning from this subroutine will effectively do
! ! Nothing to x1; it is not finalizable
! ! CALL t2f(x2)
! CALL t3f(x3)
! CALL t2f(x3%t2)
END SUBROUTINE

C.3 Clause 8 notes: The VOLATILE attribute (8.5.19)

1 The following example shows the use of a variable with the VOLATILE attribute to communicate with an asynchronous process, in this case the operating system. The program detects a user keystroke on the terminal and reacts at a convenient point in its processing.

2 The VOLATILE attribute is necessary to prevent an optimizing compiler from storing the communication variable in a register or from doing flow analysis and deciding that the EXIT statement can never be executed.

SUBROUTINE TERMINATE_ITERATIONS
LOGICAL, VOLATILE :: USER_HIT_ANY_KEY

! Have the OS start to look for a user keystroke and set the variable
! "USER_HIT_ANY_KEY" to TRUE as soon as it detects a keystroke.
! This call is operating system dependent.
CALL OS_BEGIN_DETECT_USER_KEYSTROKE( USER_HIT_ANY_KEY )
USER_HIT_ANY_KEY = .FALSE. ! This will ignore any recent keystrokes.
PRINT *, " Hit any key to terminate iterations!"
DO I = 1,100
... ! Compute a value for R.
PRINT *, I, R
IF (USER_HIT_ANY_KEY) EXIT
ENDDO

! Have the OS stop looking for user keystrokes.
CALL OS_STOP_DETECT_USER_KEYSTROKE

END SUBROUTINE TERMINATE_ITERATIONS

C.4 Clause 9 notes

C.4.1 Structure components (9.4.2)

1 Components of a structure are referenced by writing the components of successive levels of the structure hierarchy until the desired component is described. For example,

```
TYPE ID_NUMBERS
  INTEGER SSN
  INTEGER EMPLOYEE_NUMBER
END TYPE ID_NUMBERS

TYPE PERSON_ID
  CHARACTER (LEN=30) LAST_NAME
  CHARACTER (LEN=1) MIDDLE_INITIAL
  CHARACTER (LEN=30) FIRST_NAME
  TYPE (ID_NUMBERS) NUMBER
END TYPE PERSON_ID

TYPE PERSON
  INTEGER AGE
  TYPE (PERSON_ID) ID
END TYPE PERSON

TYPE (PERSON) GEORGE, MARY

PRINT *, GEORGE % AGE          ! Print the AGE component
PRINT *, MARY % ID % LAST_NAME  ! Print LAST_NAME of MARY
PRINT *, MARY % ID % NUMBER % SSN ! Print SSN of MARY
PRINT *, GEORGE % ID % NUMBER   ! Print SSN and EMPLOYEE_NUMBER of GEORGE
```

2 A structure component can be a data object of intrinsic type as in the case of GEORGE % AGE or it can be of derived type as in the case of GEORGE % ID % NUMBER. The resultant component can be a scalar or an array of intrinsic or derived type.

```
TYPE LARGE
  INTEGER ELT (10)
  INTEGER VAL
END TYPE LARGE
```
Components of an object of extensible type that are inherited from the parent type can be accessed as a whole by using the parent component name, or individually, either with or without qualifying them by the parent component name. For example:

```fortran
TYPE POINT ! A base type
  REAL :: X, Y
END TYPE POINT

TYPE, EXTENDS(POINT) :: COLOR_POINT ! An extension of TYPE(POINT)
  ! Components X and Y, and component name POINT, inherited from parent
  INTEGER :: COLOR
END TYPE COLOR_POINT

TYPE(POINT) :: PV = POINT(1.0, 2.0)
TYPE(COLOR_POINT) :: CPV = COLOR_POINT(POINT=PV, COLOR=3)

PRINT *, CPV%POINT ! Prints 1.0 and 2.0
PRINT *, CPV%POINT%X, CPV%POINT%Y ! And this does, too
PRINT *, CPV%X, CPV%Y ! And this does, too
```

### C.4.2 Allocation with dynamic type (9.7.1)

The following example illustrates the use of allocation with the value and dynamic type of the allocated object given by another object. The example copies a list of objects of any type. It copies the list starting at IN_LIST. After copying, each element of the list starting at LIST_COPY has a polymorphic component, ITEM, for which both the value and type are taken from the ITEM component of the corresponding element of the list starting at IN_LIST.

```fortran
TYPE :: LIST ! A list of anything
  TYPE(LIST), POINTER :: NEXT => NULL()
  CLASS(*), ALLOCATABLE :: ITEM
END TYPE LIST

... TYPE(LIST), POINTER :: IN_LIST, LIST_COPY => NULL()
TYPE(LIST), POINTER :: IN_WALK, NEW_TAIL
  ! Copy IN_LIST to LIST_COPY
IF (ASSOCIATED(IN_LIST)) THEN
  IN_WALK => IN_LIST
  ALLOCATE(LIST_COPY)
```
NEW_TAIL => LIST_COPY
DO
   ALLOCATE(NEW_TAIL%ITEM, SOURCE=IN_WALK%ITEM)
   IN_WALK => IN_WALK%NEXT
   IF (.NOT. ASSOCIATED(IN_WALK)) EXIT
   ALLOCATE(NEW_TAIL%NEXT)
   NEW_TAIL => NEW_TAIL%NEXT
END DO
END IF

C.5 Clause 10 notes

C.5.1 Evaluation of function references (10.1.7)
1 If more than one function reference appears in a statement, they can be executed in any order (subject to a
function result being evaluated after the evaluation of its arguments) and their values cannot depend on the order
of execution. This lack of dependence on order of evaluation enables parallel execution of the function references.

C.5.2 Pointers in expressions (10.1.9.2)
1 A pointer is considered to be like any other variable when it is used as a primary in an expression. If a pointer
is used as an operand to an operator that expects a value, the pointer will automatically deliver the value stored
in the space described by the pointer, that is, the value of the target object associated with the pointer.

C.5.3 Pointers in variable definition contexts (10.2.1.3, 19.6.7)
1 The appearance of a pointer in a context that requires its value is a reference to its target. Similarly, where a
pointer appears in a variable definition context the variable that is defined is the target of the pointer.
2 Executing the program fragment
   REAL, POINTER :: A
   REAL, TARGET :: B = 10.0
   A => B
   A = 42.0
   PRINT '(F4.1)', B
produces “42.0” as output.

C.6 Clause 11 notes

C.6.1 The SELECT CASE construct (11.1.9)
1 At most one case block is selected for execution within a SELECT CASE construct, and there is no fall-through
from one block into another block within a SELECT CASE construct. Thus there is no requirement for the user
to exit explicitly from a block.
C.6.2 Loop control (11.1.7)

1 Fortran provides several forms of loop control:

(1) With an iteration count and a DO variable. This is the classic Fortran DO loop.

(2) Test a logical condition before each execution of the loop (DO WHILE).

(3) DO “forever”.

C.6.3 Examples of DO constructs (11.1.7)

1 The following are all valid examples of DO constructs.

Example 1:

SUM = 0.0
READ (IUN) N
OUTER: DO L = 1, N ! A DO with a construct name
   READ (IUN) IQUAL, M, ARRAY (1:M)
   IF (IQUAL < IQUAL_MIN) CYCLE OUTER ! Skip inner loop
INNER: DO 40 I = 1, M ! A DO with a label and a name
      CALL CALCULATE (ARRAY (I), RESULT)
      IF (RESULT < 0.0) CYCLE
      SUM = SUM + RESULT
      IF (SUM > SUM_MAX) EXIT OUTER
40   END DO INNER
END DO OUTER

2 The outer loop has an iteration count of MAX (N, 0), and will execute that number of times or until SUM exceeds SUM_MAX, in which case the EXIT OUTER statement terminates both loops. The inner loop is skipped by the first CYCLE statement if the quality flag, IQUAL, is too low. If CALCULATE returns a negative RESULT, the second CYCLE statement prevents it from being summed. Both loops have construct names and the inner loop also has a label. A construct name is required in the EXIT statement in order to terminate both loops, but is optional in the CYCLE statements because each belongs to its innermost loop.

Example 2:

N = 0
DO 50, I = 1, 10
   J = I
   DO K = 1, 5
      L = K
      N = N + 1 ! This statement executes 50 times
   END DO ! Nonlabeled DO inside a labeled DO
50 CONTINUE

3 After execution of the above program fragment, I = 11, J = 10, K = 6, L = 5, and N = 50.

Example 3:
After execution of the above program fragment, I = 11, J = 10, K = 5, N = 0, and L is not defined by these statements.

**C.6.4 Examples of invalid DO constructs (11.1.7)**

1. The following are all examples of invalid skeleton **DO constructs**:

   **Example 1:**
   ```
   DO I = 1, 10
   ...
   END DO LOOP ! No matching construct name
   ```

   **Example 2:**
   ```
   LOOP: DO 1000 I = 1, 10 ! No matching construct name
   ...
   1000 CONTINUE
   ```

   **Example 3:**
   ```
   LOOP1: DO
   ...
   END DO LOOP2 ! Construct names don’t match
   ```

   **Example 4:**
   ```
   DO I = 1, 10 ! Label required or ...
   ...
   1010 CONTINUE ! ... END DO required
   ```

   **Example 5:**
   ```
   DO 1020 I = 1, 10
   ...
   1021 END DO ! Labels don’t match
   ```

   **Example 6:**
FIRST: DO I = 1, 10
SECOND: DO J = 1, 5
...  
END DO FIRST ! Improperly nested DOs  
END DO SECOND

C.6.5 Simple example using events

A tree is a graph in which every node except one has a single “parent” node to which it is connected by an edge. The node without a parent is the “root” of the tree. The nodes that have a particular node as their parent are the “children” of that node. The root is at level 1, its children are at level 2, and so on.

A multifrontal code to solve a sparse set of linear equations involves a tree. Work at a node can start after all of its children’s work is complete and their data have been passed to it.

Here we assume that each node has been assigned to an image. Each image has a list of its nodes and these are ordered in decreasing tree level (all those at level \( L \) preceding those at level \( L - 1 \)). For each node, array elements hold the number of children, details about the parent, and an event variable. This allows the processing to proceed asynchronously subject to the rule that a parent has to wait for all its children.

Outline of example code:

```fortran
PROGRAM TREE
  USE, INTRINSIC :: ISO_FORTRAN_ENV
  INTEGER, ALLOCATABLE :: NODE (:) ! Tree nodes that this image handles.
  INTEGER, ALLOCATABLE :: NC (:) ! NODE(I) has NC(I) children.
  INTEGER, ALLOCATABLE :: PARENT (:), SUB (:)
    ! The parent of NODE (I) is NODE (SUB (I)) [PARENT (I)].
  TYPE (EVENT_TYPE), ALLOCATABLE :: DONE (: [:]
  INTEGER :: I, J, STATUS

  ! Set up the tree, including allocation of all arrays.
  DO I = 1, SIZE (NODE)
    ! Wait for children to complete
    IF (NC (I) > 0) THEN
      EVENT WAIT (DONE (I), UNTIL_COUNT=NC (I), STAT=STATUS)
      IF (STATUS/=0) EXIT
    END IF
  END DO

  ! Process node, using data from children.
  IF (PARENT (I)>0) THEN
    ! Node is not the root.
    ! Place result on image PARENT (I) for node NODE (SUB) [PARENT (I)]
    ! Tell PARENT (I) that this has been done.
    EVENT POST (DONE (SUB (I)) [PARENT (I)], STAT=STATUS)
    IF (STATUS/=0) EXIT
  END IF
END PROGRAM TREE
```
C.6.6 Example using three teams

The following example illustrates the structure of a routine that will compute fluxes based on surface properties over land, sea, and ice, each in a different team. Each image will deal with areas containing exactly one of the three surface types.

```
SUBROUTINE COMPUTE_FLUXES (FLUX_MOM, FLUX_SENS, FLUX_LAT)
  USE, INTRINSIC :: ISO_FORTRAN_ENV, ONLY: TEAM_TYPE
  REAL, INTENT (OUT) :: FLUX_MOM (:,:), FLUX_SENS (:,:), FLUX_LAT (:,:)
  INTEGER, PARAMETER :: LAND = 1, SEA = 2, ICE = 3
  CHARACTER (LEN=10) :: SURFACE_TYPE
  INTEGER :: MY_SURFACE_TYPE, N_IMAGE
  TYPE (TEAM_TYPE) :: TEAM_SURFACE_TYPE

  CALL GET_SURFACE_TYPE(THIS_IMAGE (), SURFACE_TYPE)
  SELECT CASE (SURFACE_TYPE)
    CASE ("LAND")
      MY_SURFACE_TYPE = LAND
    CASE ("SEA")
      MY_SURFACE_TYPE = SEA
    CASE ("ICE")
      MY_SURFACE_TYPE = ICE
    CASE DEFAULT
      ERROR STOP
  END SELECT
  FORM TEAM (MY_SURFACE_TYPE, TEAM_SURFACE_TYPE)

  CHANGE TEAM (TEAM_SURFACE_TYPE)
  SELECT CASE (TEAM_NUMBER ( ))
    CASE (LAND) ! Compute fluxes over land surface
      CALL COMPUTE_FLUXES_LAND (FLUX_MOM, FLUX_SENS, FLUX_LAT)
    CASE (SEA) ! Compute fluxes over sea surface
      CALL COMPUTE_FLUXES_SEA (FLUX_MOM, FLUX_SENS, FLUX_LAT)
    CASE (ICE) ! Compute fluxes over ice surface
      CALL COMPUTE_FLUXES_ICE (FLUX_MOM, FLUX_SENS, FLUX_LAT)
    CASE DEFAULT
      ERROR STOP
  END SELECT
  END TEAM
END SUBROUTINE COMPUTE_FLUXES
```

C.6.7 Accessing coarrays in sibling teams

The following program illustrates subdividing a 4 × 4 grid into 2 × 2 teams, and the denotation of sibling teams.

```
PROGRAM DEMO
  ! Initial team : 16 images. Algorithm design is a 4 by 4 grid.
  ! Desire 4 teams, for the upper left (UL), upper right (UR),
```

ISO/IEC JTC 1/SC 22/WG5/N2118 559
USE, INTRINSIC :: ISO_FORTRAN_ENV, ONLY: TEAM_TYPE

TYPE (TEAM_TYPE) :: t

INTEGER, PARAMETER :: UL=11, UR=22, LL=33, LR=44

REAL :: A(10,10)

INTEGER :: MYPE, TEAMNUM, NEWPE

INTEGER :: UL_IMAGE_LIST(4) = [1, 2, 5, 6], k

LL_IMAGE_LIST(4) = UL_IMAGE_LIST + 2, k

UR_IMAGE_LIST(4) = UL_IMAGE_LIST + 8, k

LR_IMAGE_LIST(4) = UL_IMAGE_LIST + 10

MYPE = THIS_IMAGE()

IF (ANY (MYPE==UL_IMAGE_LIST)) TEAMNUM = UL

IF (ANY (MYPE==LL_IMAGE_LIST)) TEAMNUM = LL

IF (ANY (MYPE==UR_IMAGE_LIST)) TEAMNUM = UR

IF (ANY (MYPE==LR_IMAGE_LIST)) TEAMNUM = LR

FORM TEAM (TEAMNUM, T)

A = 3.14

CHANGE TEAM (T, B[2,*] => A)

! Inside change team, image pattern for B is a 2 by 2 grid.

B (5, 5) = B (1, 1)[2, 1]

! Outside the team addressing:

NEWPE = THIS_IMAGE()

SELECT CASE (TEAM_NUMBER ())

CASE (UL)

IF (NEWPE==3) THEN

! Right column of UL gets left column of UR.

B (:, 10) = B (:, 1)[1, 1, TEAM_NUMBER=UR]

ELSE IF (NEWPE==4) THEN

B (:, 10) = B (:, 1)[2, 1, TEAM_NUMBER=UR]

END IF

CASE (LL)

! Similar to complete column exchange across middle of the original grid.

...

END SELECT

END TEAM

END PROGRAM DEMO

C.6.8 Example involving failed images

Parallel algorithms often use work sharing schemes based on a specific mapping between image indices and global data addressing. To allow such programs to continue when one or more images fail, spare images can be used...
to re-establish execution of the algorithm with the failed images replaced by spare images, while retaining the
previous image mapping for nonfailed images.

The following example illustrates how this might be done. In this example, failure cannot be tolerated for image
1 in the initial team.

```fortran
PROGRAM possibly_recoverable_simulation
USE, INTRINSIC :: ISO_FORTRAN_ENV, ONLY: TEAM_TYPE, STAT_FAILED_IMAGE
IMPLICIT NONE
INTEGER, ALLOCATABLE :: failures (:)
INTEGER :: images_used, i, images_spare, status
INTEGER :: id [*], me [*]
TYPE (TEAM_TYPE) :: simulation_team
LOGICAL :: read_checkpoint, done [*]

! Keep 1% spare images if we have a lot, just 1 if 10-199 images, 0 if <10.
images_spare = MAX (INT (0.01*NUM_IMAGES ()), 0, MIN (NUM_IMAGES () - 10, 1)
images_used = NUM_IMAGES () - images_spare
read_checkpoint = THIS_IMAGE () > images_used

setup : DO
me = THIS_IMAGE ()
id = MERGE (1, 2, me<=images_used)
!
! Set up spare images as replacement for failed ones.
!
IF (IMAGE_STATUS (1) == STAT_FAILED_IMAGE) &
   ERROR STOP "cannot recover"
IF (me == 1) THEN
   failures = FAILED IMAGES ()
k = images_used
DO i = 1, SIZE (failures)
   DO k = k+1, NUM_IMAGES ()
      IF (IMAGE_STATUS (k) == 0) EXIT
   END DO
   IF (k > NUM_IMAGES ()) ERROR STOP "cannot recover"
   me [k] = failures (i)
id [k] = 1
END DO
images_used = k
END IF
!
! Set up a simulation team of constant size.
! Team 2 is the set of spares, so does not participate in the simulation.
!
FORM TEAM (id, simulation_team, NEW_INDEX=me, STAT=status)
simulation : CHANGE TEAM (simulation_team, STAT=status)
```
IF (status == STAT_FAILED_IMAGE) EXIT simulation
IF (TEAM_NUMBER () == 1) THEN
iter : DO
   CALL simulation_procedure (read_checkpoint, status, done)
   ! The simulation_procedure:
   ! - sets up and performs some part of the simulation;
   ! - resets to the last checkpoint if requested;
   ! - sets status from its internal synchronizations;
   ! - sets done to .TRUE. when the simulation has completed.
   IF (status == STAT_FAILED_IMAGE) THEN
      read_checkpoint = .TRUE.
      EXIT simulation
   ELSE IF (done)
      EXIT iter
   END IF
   read_checkpoint = .FALSE.
END DO iter
END IF
END TEAM simulation (STAT=status)
SYNC ALL (STAT=status)
IF (THIS_IMAGE () > images_used) done = done[1]
IF (done) EXIT setup
END DO setup
END PROGRAM possibly_recoverable_simulation

3 Supporting fault-tolerant execution imposes obligations on library writers who use the parallel language facilities. Every synchronization statement, allocation or deallocation of coarrays, or invocation of a collective procedure will need to be prepared to handle error conditions, and implicit deallocation of coarrays will need to be avoided. Also, coarray module variables that are allocated inside the team execution context are not persistent.

C.6.9 EVENT_QUERY example that tolerates image failure

1 This example is an adaptation of the later EVENT_QUERY example of C.11.2 to make it able to execute in the presence of the failure of one or more of the worker images. The function create_work_item now accepts an integer argument to indicate which work item is required. It is assumed that the work items are indexed 1, 2, ... . It is also assumed that if an image fails while processing a work item, that work item can subsequently be processed by another image.

PROGRAM work_share
USE, INTRINSIC :: ISO_FORTRAN_ENV, ONLY: EVENT_TYPE
USE :: mod_work, ONLY: & ! Module that creates work items
   work, & ! Type for holding a work item
   create_work_item, & ! Function that creates work item
   process_item, & ! Function that processes an item
   work_done & ! Logical function that returns true
              ! if all work done

562 ISO/IEC JTC 1/SC 22/WG5/N2118
TYPE :: worker_type
  TYPE (EVENT_TYPE), ALLOCATABLE :: free (:)
END TYPE

TYPE (EVENT_TYPE) :: submit [*] ! Whether work ready for a worker
TYPE (worker_type) :: worker [*] ! Whether worker is free
TYPE (work) :: work_item [*] ! Holds the data for a work item
INTEGER :: count, i, k, kk, nbusy [*], np, status
INTEGER, ALLOCATABLE :: working (:) ! Items being worked on
INTEGER, ALLOCATABLE :: pending (:) ! Items pending after image failure

IF (THIS_IMAGE () == 1) THEN
  ! Get started
  ALLOCATE (worker%free (2:NUM_IMAGES ()))
  ALLOCATE (working (2: NUM_IMAGES ()), pending(NUM_IMAGES ()-1))
  nbusy = 0 ! This holds the number of workers working
  k = 1 ! Index of next work item
  np = 0 ! Number of work items in array pending
  DO i = 2, NUM_IMAGES () ! Start the workers working
     IF (work_done ()) EXIT
     working (i) = 0
     IF (IMAGE_STATUS (i) == STAT_FAILED_IMAGE) CYCLE
     work_item [i] = create_work_item (k)
     working (i) = k
     k = k + 1
     nbusy = nbusy + 1
     EVENT POST (submit [i], STAT=status)
  END DO
  ! Main work distribution loop
  master : DO
     image : DO i = 2, NUM_IMAGES ()
        IF (IMAGE_STATUS (i) == STAT_FAILED_IMAGE) THEN
           IF (working (i)>0) THEN ! It failed while working
              np = np + 1
              pending (np) = working (i)
              working (i) = 0
           END IF
        END IF
        CALL EVENT_QUERY (worker%free (i), count)
        IF (count == 0) CYCLE image ! Worker is not free
        EVENT WAIT (worker%free (i))
        nbusy = nbusy - 1
        IF (np>0) THEN
           kk = pending (np)
           np = np - 1
        ELSE
IF (work_done ()) CYCLE image

kk = k
k = k + 1
END IF

nbusy = nbusy + 1
working (i) = kk
work_item [i] = create_work_item (kk)
EVENT POST (submit [i], STAT=status)

! If image i has failed, the failure will be handled on
! the next iteration of the master loop.

END DO image

IF ( nbusy==0 ) THEN ! All done. Exit on all images.
  DO i = 2, NUM_IMAGES ()
    EVENT POST (submit [i], STAT=status)
    IF (status == STAT_FAILED_IMAGE) CYCLE
  END DO
  EXIT master
END IF
END DO master

ELSE
  ! Work processing loop
  worker : DO
    EVENT WAIT (submit)
    IF (nbusy [1] == 0) EXIT worker
    CALL process_item(work_item)
    EVENT POST (worker[1]%free (THIS_IMAGE ()))
  END DO worker
  END IF
END PROGRAM work_share

C.7 Clause 12 notes

C.7.1 External files (12.3)

This document accommodates, but does not require, file cataloging. To do this, several concepts are introduced.

C.7.1.1 File existence (12.3.2)

Totally independent of the connection state is the property of existence, this being a file property. The processor
“knows” of a set of files that exist at a given time for a given program. This set would include tapes ready to
read, files in a catalog, a keyboard, a printer, etc. The set might exclude files inaccessible to the program because
of security, because they are already in use by another program, etc. This document does not specify which
files exist, hence wide latitude is available to a processor to implement security, locks, privilege techniques, etc.
Existence is a convenient concept to designate all of the files that a program can potentially process.
All four combinations of connection and existence can occur:

<table>
<thead>
<tr>
<th>Connect</th>
<th>Exist</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>A card reader loaded and ready to be read</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>A printer before the first line is written</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>A file named 'JOAN' in the catalog</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>A file on a reel of tape, not known to the processor</td>
</tr>
</tbody>
</table>

Means are provided to create, delete, connect, and disconnect files.

C.7.1.2 File access (12.3.3)

This document does not address problems of security, protection, locking, and many other concepts that might be part of the concept of “right of access”. Such concepts are considered to be in the province of an operating system.

The OPEN and INQUIRE statements can be extended naturally to consider these things.

Possible access methods for a file are: sequential, stream and direct. The processor might implement three different types of files, each with its own access method. It might instead implement one type of file with three different access methods.

Direct access to files is of a simple and commonly available type, that is, fixed-length records. The key is a positive integer.

C.7.1.3 File connection (12.5)

Before any input/output can be performed on a file, it needs to be connected to a unit. The unit then serves as a designator for that file as long as it is connected. To be connected does not imply that “buffers” have or have not been allocated, that “file-control tables” have or have not been filled, or that any other method of implementation has been used. Connection means that (barring some other fault) a READ or WRITE statement can be executed on the unit, hence on the file. Without a connection, a READ or WRITE statement cannot be executed.

C.7.1.4 File names (12.5.6.10)

A file can have a name. The form of a file name is not specified. If a system does not have some form of cataloging or tape labeling for at least some of its files, all file names disappear at the termination of execution. This is a valid implementation. Nowhere does this document require names to survive for any period of time longer than the execution time span of a program. Therefore, this document does not impose cataloging as a prerequisite. The naming feature is intended to enable use of a cataloging system where one exists.

C.7.2 Nonadvancing input/output (12.3.4.2)

Data transfer statements affect the positioning of an external file. In Fortran 77, if no error or end-of-file condition exists, the file is positioned after the record just read or written and that record becomes the preceding record. This document contains the ADVANCE= specifier in a data transfer statement that provides the capability of maintaining a position within the current record from one formatted data transfer statement to the next data transfer statement. The value NO provides this capability. The value YES positions the file after the record just read or written. The default is YES.
The tab edit descriptor and the slash are still appropriate for use with this type of record access but the tab cannot reposition before the left tab limit.

A BACKSPACE of a file that is positioned within a record causes the specified unit to be positioned before the current record.

If the next input/output operation on a file after a nonadvancing write is a rewind, backspace, end file or close operation, the file is positioned implicitly after the current record before an ENDFILE record is written to the file, that is, a REWIND, BACKSPACE, or ENDFILE statement following a nonadvancing WRITE statement causes the file to be positioned at the end of the current output record before the endfile record is written to the file.

This document provides a SIZE= specifier to be used with formatted data transfer statements. The variable in the SIZE= specifier is assigned the count of the number of characters that make up the sequence of values read by the data edit descriptors in the input statement.

The count is especially helpful if there is only one list item in the input list because it is the number of characters that appeared for the item.

The EOR= specifier is provided to indicate when an EOR condition is encountered during a nonadvancing data transfer statement. The EOR condition is not an error condition. If this specifier appears, an input list item that requires more characters than the record contained is padded with blanks if PAD= 'YES' is in effect. This means that the input list item completed successfully. The file is positioned after the current record. If the IOSTAT= specifier appears, the specified variable is defined with the value of the named constant IOSTAT_EOR from the intrinsic module ISO_FORTRAN_ENV and the data transfer statement is terminated. Program execution continues with the statement specified in the EOR= specifier. The EOR= specifier gives the capability of taking control of execution when the EOR condition is encountered. The do-variables in io-implied-dos retain their last defined value and any remaining items in the input-item-list retain their definition status when an EOR condition occurs. If the SIZE= specifier appears, the specified variable is assigned the number of characters read with the data edit descriptors during the READ statement.

For nonadvancing input, the processor is not required to read partial records. The processor could read the entire record into an internal buffer and make successive portions of the record available to successive input statements.

In an implementation of nonadvancing input/output in which a nonadvancing write to a terminal device causes immediate display of the output, such a write can be used as a mechanism to output a prompt. In this case, the statement

```
WRITE (*, FMT='(A)', ADVANCE='NO') 'CONTINUE?(Y/N): '  
```

would result in the prompt

```
CONTINUE?(Y/N):  
```

being displayed with no subsequent line feed.

The response, which might be read by a statement of the form

```
READ (*, FMT='(A)') ANSWER  
```

can then be entered on the same line as the prompt as in

```
CONTINUE?(Y/N): Y  
```
This document does not require that an implementation of nonadvancing input/output operate in this manner.
For example, an implementation of nonadvancing output in which the display of the output is deferred until the current record is complete is also standard-conforming. Such an implementation will not, however, allow a prompting mechanism of this kind to operate.

C.7.3 OPEN statement (12.5.6)

A file can become connected to a unit either by preconnection or by execution of an OPEN statement. Preconnection is performed prior to the beginning of execution of a program by means external to Fortran. For example, it could be done by job control action or by processor-established defaults. Execution of an OPEN statement is not required in order to access preconnected files (12.5.5).

The OPEN statement provides a means to access existing files that are not preconnected. An OPEN statement can be used in either of two ways: with a file name (open-by-name) and without a file name (open-by-unit). A unit is given in either case. Open-by-name connects the specified file to the specified unit. Open-by-unit connects a processor-dependent default file to the specified unit. (The default file might or might not have a name.)

Therefore, there are three ways a file can become connected and hence processed: preconnection, open-by-name, and open-by-unit. Once a file is connected, there is no means in standard Fortran to determine how it became connected.

An OPEN statement can also be used to create a new file. In fact, any of the foregoing three connection methods can be performed on a file that does not exist. When a unit is preconnected, writing the first record creates the file. With the other two methods, execution of the OPEN statement creates the file.

When an OPEN statement is executed, the unit specified in the OPEN statement might or might not already be connected to a file. If it is already connected to a file (either through preconnection or by prior execution of an OPEN statement), then omitting the FILE= specifier in the OPEN statement implies that the file is to remain connected to the unit. Such an OPEN statement can be used to change the values of the blank interpretation mode, decimal edit mode, pad mode, input/output rounding mode, delimiter mode, and sign mode.

If the value of the ACTION= specifier is WRITE, then a READ statement cannot refer to the connection. ACTION = 'WRITE' does not restrict positioning by a BACKSPACE statement or positioning specified by the POSITION= specifier with the value APPEND. However, a BACKSPACE statement or an OPEN statement containing POSITION = 'APPEND' might fail if the processor needs to read the file to achieve the positioning.

The following examples illustrate these rules. In the first example, unit 10 is preconnected to a SCRATCH file; the OPEN statement changes the value of PAD= to YES.

```
CHARACTER (LEN = 20) CH1
WRITE (10, '(A)') 'THIS IS RECORD 1'
OPEN (UNIT = 10, STATUS = 'OLD', PAD = 'YES')
REWIND 10
READ (10, '(A20)') CH1 ! CH1 now has the value
                   ! 'THIS IS RECORD 1 '
```

In the next example, unit 12 is first connected to a file named FRED, with a status of OLD. The second OPEN statement then opens unit 12 again, retaining the connection to the file FRED, but changing the value of the DELIM= specifier to QUOTE.
CHARACTER (LEN = 25) CH2, CH3
OPEN (12, FILE = 'FRED', STATUS = 'OLD', DELIM = 'NONE')
CH2 = '"THIS STRING HAS QUOTES."'
! Quotes in string CH2
WRITE (12, *) CH2 ! Written with no delimiters
OPEN (12, DELIM = 'QUOTE') ! Now quote is the delimiter
REWIND 12
READ (12, *) CH3 ! CH3 now has the value
! 'THIS STRING HAS QUOTES. '

The next example is invalid because it attempts to change the value of the STATUS= specifier.

OPEN (10, FILE = 'FRED', STATUS = 'OLD')
WRITE (10, *) A, B, C
OPEN (10, STATUS = 'SCRATCH') ! Attempts to make FRED a SCRATCH file

The previous example could be made valid by closing the unit first, as in the next example.

OPEN (10, FILE = 'FRED', STATUS = 'OLD')
WRITE (10, *) A, B, C
CLOSE (10)
OPEN (10, STATUS = 'SCRATCH') ! Opens a different SCRATCH file

C.7.4 Connection properties (12.5.4)

When a unit becomes connected to a file, either by execution of an OPEN statement or by preconnection, the following connection properties, among others, are established.

1. An access method, which is sequential, direct, or stream, is established for the connection (12.5.6.3).
2. A form, which is formatted or unformatted, is established for a connection to a file that exists or is created by the connection. For a connection that results from execution of an OPEN statement, a default form (which depends on the access method, as described in 12.3.3) is established if no form is specified. For a preconnected file that exists, a form is established by preconnection. For a preconnected file that does not exist, a form might be established, or the establishment of a form might be delayed until the file is created (for example, by execution of a formatted or unformatted WRITE statement) (12.5.6.11).
3. A record length might be established. If the access method is direct, the connection establishes a record length that specifies the length of each record of the file. A direct access file can only contain records that are all of equal length.
4. A sequential file can contain records of varying lengths. In this case, the record length established specifies the maximum length of a record in the file (12.5.6.15).

A processor has wide latitude in adapting these concepts and actions to its own cataloging and job control conventions. Some processors might need job control action to specify the set of files that exist or that will be created by a program. Some processors might not need any job control action prior to execution. This document enables processors to perform dynamic open, close, or file creation operations, but it does not require such capabilities of the processor.
3 The meaning of “open” in contexts other than Fortran might include such things as mounting a tape, console messages, spooling, label checking, security checking, etc. These actions might occur upon job control action external to Fortran, upon execution of an OPEN statement, or upon execution of the first read or write of the file. The OPEN statement describes properties of the connection to the file and might or might not cause physical activities to take place.

C.7.5 Asynchronous input/output (12.6.2.5)

1 Rather than limit support for asynchronous input/output to what has been traditionally provided by facilities such as BUFFERIN/BUFFEROUT, this document builds upon existing Fortran syntax. This permits alternative approaches for implementing asynchronous input/output, and simplifies the task of adapting existing standard-conforming programs to use asynchronous input/output.

2 Not all processors actually perform input/output asynchronously, nor will every processor that does be able to handle data transfer statements with complicated input/output item lists in an asynchronous manner. Such processors can still be standard-conforming.

3 This document allows for at least two different conceptual models for asynchronous input/output.

4 Model 1: the processor performs asynchronous input/output when the item list is simple (perhaps one contiguous named array) and the input/output is unformatted. The implementation cost is reduced, and this is the scenario most likely to be beneficial on traditional “big-iron” machines.

5 Model 2: The processor is free to do any of the following:

   (1) on output, create a buffer inside the input/output library, completely formatted, and then start an asynchronous write of the buffer, and immediately return to the next statement in the program. The processor is free to wait for previously issued WRITEs, or not, or

   (2) pass the input/output list addresses to another processor/process, which processes the list items independently of the processor that executes the user’s code. The addresses of the list items will need to be computed before the asynchronous READ/WRITE statement completes. There is still an ordering requirement on list item processing to handle things like READ (...) N,(a(i),i=1,N).

6 A program can issue a large number of asynchronous input/output requests, without waiting for any of them to complete, and then wait for any or all of them. That does not constitute a requirement for the processor to keep track of each individual request separately.

7 It is not necessary for all requests to be tracked by the runtime library. If an ID= specifier does not appear in on a READ or WRITE statement, the runtime library can forget about this particular request once it has successfully completed. If an error or end-of-file condition occurs for a request, the processor can report this during any input/output operation to that unit. If an ID= specifier appears, the processor’s runtime input/output library will need to keep track of any end-of-file or error conditions for that particular input/output request. However, if the input/output request succeeds without any exceptional conditions occurring, then the runtime can forget that ID= value. A runtime library might only keep track of the last request made, or perhaps a very few. Then, when a user WAITs for a particular request, either the library will know about it (and does the right thing with respect to error handling, etc.), or can assume it is a request that successfully completed and was forgotten about (and will just return without signaling any end-of-file or error condition). A standard-conforming program can only pass valid ID= values, but there is no requirement on the processor to detect invalid ID= values. There might be a processor dependent limit on how many outstanding input/output requests that generate an end-of-file or
error condition can be handled before the processor runs out of memory to keep track of such conditions. The restrictions on the SIZE= variables are designed to enable the processor to update such variables at any time (after the request has been processed, but before the wait operation), and then forget about them. Only error and end-of-file conditions are expected to be tracked by individual request by the runtime, and then only if an ID= specifier appears. The END= and EOR= specifiers have not been added to all statements that can perform wait operations. Instead, the IOSTAT variable can be queried after a wait operation to handle this situation. This choice was made because the WAIT statement is expected to be the usual method of waiting for input/output to complete (and WAIT does support the END= and EOR= specifiers). This particular choice is philosophical, and was not based on significant technical difficulties.

The requirement to set the IOSTAT variable correctly means that a processor will need to remember which input/output requests encountered an end-of-record condition, so that a subsequent wait operation can return the correct IOSTAT value. Therefore there might be a processor defined limit on the number of outstanding nonadvancing input/output requests that have encountered an end-of-record condition (constrained by available memory to keep track of this information, similar to end-of-file and error conditions).

C.8 Clause 13 notes

C.8.1 Number of records (13.4, 13.5, 13.8.2)

The number of records read by an explicitly formatted advancing input statement can be determined from the following rule: a record is read at the beginning of the format scan (even if the input list is empty unless the most recently previous operation on the unit was not a nonadvancing read operation), at each slash edit descriptor encountered in the format, and when a format rescan occurs at the end of the format.

The number of records written by an explicitly formatted advancing output statement can be determined from the following rule: a record is written when a slash edit descriptor is encountered in the format, when a format rescan occurs at the end of the format, and at completion of execution of an advancing output statement (even if the output list is empty). Thus, the occurrence of \( n \) successive slashes between two other edit descriptors causes \( n - 1 \) blank lines if the records are printed. The occurrence of \( n \) slashes at the beginning or end of a complete format specification causes \( n \) blank lines if the records are printed. However, a complete format specification containing \( n \) slashes (\( n > 0 \)) and no other edit descriptors causes \( n + 1 \) blank lines if the records are printed. For example, the statements

\[
\text{PRINT 3}
\]
\[
3 \text{ FORMAT (/)}
\]

will write two records that cause two blank lines if the records are printed.

C.8.2 List-directed input (13.10.3)

The following examples illustrate list-directed input. A blank character is represented by b.

Example 1:

Program:

\[
\text{J = 3}
\]
\[
\text{READ *, I}
\]
\[
\text{READ *, J}
\]
Sequential input file:

record 1: b1b,4bbbbbb
record 2: ,2bbbbbbbb

Result: I = 1, J = 3.

Explanation: The second READ statement reads the second record. The initial comma in the record designates a null value; therefore, J is not redefined.

Example 2:

Program:

CHARACTER A *8, B *1
READ *, A, B

Sequential input file:

record 1: 'bbbbbbbb'
record 2: 'QXY'b'Z'

Result: A = 'bbbbbbbb', B = 'Q'

Explanation: In the first record, the rightmost apostrophe is interpreted as delimiting the constant (it cannot be the first of a pair of embedded apostrophes representing a single apostrophe because this would involve the prohibited “splitting” of the pair by the end of a record); therefore, A is assigned the character constant 'bbbbbbbb'. The end of a record acts as a blank, which in this case is a value separator because it occurs between two constants.

C.9 Clause 14 notes

C.9.1 Main program and block data program unit (14.1, 14.3)

1 The name of the main program or of a block data program unit has no explicit use within the Fortran language. It is available for documentation and for possible use by a processor.

2 A processor might implement an unnamed program unit by assigning it a global identifier that is not used elsewhere in the program. This could be done by using a default name that does not satisfy the rules for Fortran names.

C.9.2 Dependent compilation (14.2)

1 This document, like its predecessors, is intended to enable the implementation of conforming processors in which a program can be broken into multiple units, each of which can be separately translated in preparation for execution. Such processors are commonly described as supporting separate compilation. There is an important difference between the way separate compilation can be implemented under this document and the way it could be implemented under the Fortran 77 International Standard. Under the Fortran 77 standard, any information
required to translate a program unit was specified in that program unit. Each translation was thus totally independent of all others. Under this document, a program unit can use information that was specified in a separate module and thus can be dependent on that module. The implementation of this dependency in a processor might be that the translation of a program unit depends on the results of translating one or more modules. Processors implementing the dependency this way are commonly described as supporting dependent compilation.

The dependencies involved here are new only in the sense that the Fortran processor is now aware of them. The same information dependencies existed under the Fortran 77 International Standard, but it was the programmer’s responsibility to transport the information necessary to resolve them by making redundant specifications of the information in multiple program units. The availability of separate but dependent compilation offers several potential advantages over the redundant textual specification of information.

1. Specifying information at a single place in the program ensures that different program units using that information are translated consistently. Redundant specification leaves the possibility that different information can be erroneously be specified. Even if an INCLUDE line is used to ensure that the text of the specifications is identical in all involved program units, the presence of other specifications (for example, an IMPLICIT statement) could change the interpretation of that text.

2. During the revision of a program, it is possible for a processor to assist in determining whether different program units have been translated using different (incompatible) versions of a module, although there is no requirement that a processor provide such assistance. Inconsistencies in redundant textual specification of information, on the other hand, tend to be much more difficult to detect.

3. Putting information in a module provides a way of packaging it. Without modules, redundant specifications frequently are interleaved with other specifications in a program unit, making convenient packaging of such information difficult.

4. Because a processor can be implemented such that the specifications in a module are translated once and then repeatedly referenced, there is the potential for greater efficiency than when the processor translates redundant specifications of information in multiple program units.

The exact meaning of the requirement that the public portions of a module be available at the time of reference is processor dependent. For example, a processor could consider a module to be available only after it has been compiled and require that if the module has been compiled separately, the result of that compilation be identified to the compiler when compiling program units that use it.

C.9.2.1 USE statement and dependent compilation (14.2.2)

Another benefit of the USE statement is its enhanced facilities for name management. If one needs to use only selected entities in a module, one can do so without having to worry about the names of all the other entities in that module. If one needs to use two different modules that happen to contain entities with the same name, there are several ways to deal with the conflict. If none of the entities with the same name are to be used, they can simply be ignored. If the name happens to refer to the same entity in both modules (for example, if both modules obtained it from a third module), then there is no confusion about what the name denotes and the name can be freely used. If the entities are different and one or both is to be used, the local renaming facility in the USE statement makes it possible to give those entities different names in the program unit containing the USE statements.

A benefit of using the ONLY option consistently, as compared to USE without it, is that the module from which each accessed entity is accessed is explicitly specified in each program unit. This means that one need not search
other program units to find where each one is defined. This reduces maintenance costs.

A typical implementation of dependent but separate compilation might involve storing the result of translating a module in a file whose name is derived from the name of the module. Note, however, that the name of a module is limited only by the Fortran rules and not by the names allowed in the file system. Thus the processor might have to provide a mapping between Fortran names and file system names.

The result of translating a module could reasonably either contain only the information textually specified in the module (with “pointers” to information originally textually specified in other modules) or contain all information specified in the module (including copies of information originally specified in other modules). Although the former approach would appear to save on storage space, the latter approach can greatly simplify the logic necessary to process a USE statement and can avoid the necessity of imposing a limit on the logical “nesting” of modules via the USE statement.

There is an increased potential for undetected errors in a scoping unit that uses both implicit typing and the USE statement. For example, in the program fragment

```fortran
SUBROUTINE SUB
  USE MY_MODULE
  IMPLICIT INTEGER (I-N), REAL (A-H, O-Z)
  X = F (B)
  A = G (X) + H (X + 1)
END SUBROUTINE SUB
```

X could be either an implicitly typed real variable or a variable obtained from the module MY_MODULE and might change from one to the other because of changes in MY_MODULE unrelated to the action performed by SUB. Logic errors resulting from this kind of situation can be extremely difficult to locate. Thus, the use of these features together is discouraged.

C.9.2.2 Accessibility attributes (8.5.2)

The PUBLIC and PRIVATE attributes, which can be declared only in modules, divide the entities in a module into those that are actually relevant to a scoping unit referencing the module and those that are not. This information might be used to improve the performance of a Fortran processor. For example, it might be possible to discard much of the information about the private entities once a module has been translated, thus saving on both storage and the time to search it. Similarly, it might be possible to recognize that two versions of a module differ only in the private entities they contain and avoid retranslating program units that use that module when switching from one version of the module to the other.

C.9.3 Examples of the use of modules (14.2.1)

C.9.3.1 Global data (14.2.1)

A module could contain only data objects, for example:

```fortran
MODULE DATA_MODULE
  SAVE
  REAL A (10), B, C (20,20)
  INTEGER :: I=0
  INTEGER, PARAMETER :: J=10
END MODULE DATA_MODULE
```
Data objects made global in this manner can have any combination of data types.

Access to some of these can be made by a USE statement with the ONLY option, such as:

USE DATA_MODULE, ONLY: A, B, D

and access to all of them can be made by the following USE statement:

USE DATA_MODULE

Access to all of them with some renaming to avoid name conflicts can be made by, for example:

USE DATA_MODULE, AMODULE => A, DMODULE => D

C.9.3.2 Derived types (14.2.1)

A derived type can be defined in a module and accessed in a number of program units. For example,

MODULE SPARSE
    TYPE NONZERO
        REAL A
        INTEGER I, J
    END TYPE NONZERO
END MODULE SPARSE

defines a type consisting of a real component and two integer components for holding the numerical value of a nonzero matrix element and its row and column indices.

C.9.3.3 Global allocatable arrays (14.2.1)

Many programs need large global allocatable arrays whose sizes are not known before program execution. A simple form for such a program is:

PROGRAM GLOBAL_WORK
    CALL CONFIGURE_ARRAYS ! Perform the appropriate allocations
    CALL COMPUTE ! Use the arrays in computations
END PROGRAM GLOBAL_WORK

MODULE WORK_ARRAYS ! An example set of work arrays
    INTEGER N
    REAL, ALLOCATABLE :: A (:), B (:, :), C (:, :, :)
END MODULE WORK_ARRAYS

SUBROUTINE CONFIGURE_ARRAYS ! Process to set up work arrays
    USE WORK_ARRAYS
    READ (*, *) N
    ALLOCATE (A (N), B (N, N), C (N, N, 2 * N))
END SUBROUTINE CONFIGURE_ARRAYS

SUBROUTINE COMPUTE
USE WORK_ARRAYS

... ! Computations involving arrays A, B, and C
END SUBROUTINE COMPUTE

2 Typically, many subprograms need access to the work arrays, and all such subprograms would contain the statement

USE WORK_ARRAYS

C.9.3.4 Procedure libraries (14.2.2)

1 Interface bodies for external procedures in a library can be gathered into a module. An interface body specifies an explicit interface (15.4.2.2).

2 An example is the following library module:

MODULE LIBRARY_LLS
  INTERFACE
    SUBROUTINE LLS (X, A, F, FLAG)
    REAL X (:, :)
    ! The SIZE in the next statement is an intrinsic function
    REAL, DIMENSION (SIZE (X, 2)) :: A, F
    INTEGER FLAG
    END SUBROUTINE LLS
  ... END INTERFACE
  ... END MODULE LIBRARY_LLS

3 This module provides an explicit interface that is necessary for the subroutine LLS to be invoked. For example:

USE LIBRARY_LLS

... CALL LLS (X = ABC, A = D, F = XX, FLAG = IFLAG)
... 

4 Because dummy argument names in an interface body for an external procedure are not required to be the same as in the procedure definition, different versions can be constructed for different applications using argument keywords appropriate to each application.

C.9.3.5 Operator extensions (14.2.2)

1 In order to extend an intrinsic operator symbol to have an additional meaning, an interface block specifying that operator symbol in the OPERATOR option of the INTERFACE statement could be placed in a module.

2 For example, // can be extended to perform concatenation of two derived-type objects serving as varying length character strings and + can be extended to specify matrix addition for type MATRIX or interval arithmetic addition for type INTERVAL.
A module might contain several such interface blocks. An operator can be defined by an external function (either in Fortran or some other language) and its procedure interface placed in the module.

C.9.3.6 Data abstraction (14.2.2)

In addition to providing a portable means of avoiding the redundant specification of information in multiple program units, a module provides a convenient means of “packaging” related entities, such as the definitions of the representation and operations of an abstract data type. The following example of a module defines a data abstraction for a SET type where the elements of each set are of type integer. The usual set operations of UNION, INTERSECTION, and DIFFERENCE are provided. The CARDINALITY function returns the cardinality of (number of elements in) its set argument. Two functions returning logical values are included, ELEMENT and SUBSET. ELEMENT defines the operator .IN. and SUBSET extends the operator <=. ELEMENT determines if a given scalar integer value is an element of a given set, and SUBSET determines if a given set is a subset of another given set. (Two sets can be checked for equality by comparing cardinality and checking that one is a subset of the other, or checking to see if each is a subset of the other.)

The transfer function SETF converts a vector of integer values to the corresponding set, with duplicate values removed. Thus, a vector of constant values can be used as set constants. An inverse transfer function VECTOR returns the elements of a set as a vector of values in ascending order. In this SET implementation, set data objects have a maximum cardinality of 200.

Here is the example module:

```fortran
MODULE INTEGER_SETS
  ! This module is intended to illustrate use of the module facility
  ! to define a new type, along with suitable operators.

  INTEGER, PARAMETER :: MAX_SET_CARD = 200

  TYPE SET
    ! Define SET type
    PRIVATE
    INTEGER CARD
    INTEGER ELEMENT (MAX_SET_CARD)
  END TYPE SET

  INTERFACE OPERATOR (.IN.)
    MODULE PROCEDURE ELEMENT
  END INTERFACE OPERATOR (.IN.)

  INTERFACE OPERATOR (<=)
    MODULE PROCEDURE SUBSET
  END INTERFACE OPERATOR (<=)

  INTERFACE OPERATOR (+)
    MODULE PROCEDURE UNION
  END INTERFACE OPERATOR (+)

  INTERFACE OPERATOR (-)
```

ISO/IEC JTC 1/SC 22/WG5/N2118
MODULE PROCEDURE DIFFERENCE
END INTERFACE OPERATOR (-)

INTERFACE OPERATOR (*)
MODULE PROCEDURE INTERSECTION
END INTERFACE OPERATOR (*)

CONTAINS

INTEGER FUNCTION CARDINALITY (A) ! Returns cardinality of set A
    TYPE (SET), INTENT (IN) :: A
    CARDINALITY = A % CARD
END FUNCTION CARDINALITY

LOGICAL FUNCTION ELEMENT (X, A) ! Determines if
    INTEGER, INTENT(IN) :: X ! element X is in set A
    TYPE (SET), INTENT(IN) :: A
    ELEMENT = ANY (A % ELEMENT (1 : A % CARD) == X)
END FUNCTION ELEMENT

FUNCTION UNION (A, B) ! Union of sets A and B
    TYPE (SET) UNION
    TYPE (SET), INTENT(IN) :: A, B
    INTEGER J
    UNION = A
    DO J = 1, B % CARD
        IF (.NOT. (B % ELEMENT (J) .IN. A)) THEN
            IF (UNION % CARD < MAX_SET_CARD) THEN
                UNION % CARD = UNION % CARD + 1
                UNION % ELEMENT (UNION % CARD) = B % ELEMENT (J)
            ELSE
                ! Maximum set size exceeded . . .
            END IF
        END IF
    END DO
END FUNCTION UNION

FUNCTION DIFFERENCE (A, B) ! Difference of sets A and B
    TYPE (SET) DIFFERENCE
    TYPE (SET), INTENT(IN) :: A, B
    INTEGER J, X
    DIFFERENCE % CARD = 0 ! The empty set
    DO J = 1, A % CARD
        X = A % ELEMENT (J)
        IF (.NOT. (X .IN. B)) DIFFERENCE = DIFFERENCE + SET (1, X)
    END DO
END FUNCTION DIFFERENCE

FUNCTION INTERSECTION (A, B) ! Intersection of sets A and B
  TYPE (SET), INTENT(IN) :: A, B
  INTERSECTION = A - (A - B)
END FUNCTION INTERSECTION

LOGICAL FUNCTION SUBSET (A, B) ! Determines if set A is a subset of set B
  INTEGER I
  INTEGER, INTENT(IN) :: A, B
  SUBSET = A % CARD <= B % CARD
  IF (.NOT. SUBSET) RETURN ! For efficiency
  DO I = 1, A % CARD
    SUBSET = SUBSET .AND. (A % ELEMENT (I) .IN. B)
  END DO
END FUNCTION SUBSET

TYPE (SET) FUNCTION SETF (V) ! Transfer function between a vector of elements and a set of elements
  INTEGER V (:)
  INTEGER J
  SETF % CARD = 0
  DO J = 1, SIZE (V)
    IF (.NOT. (V (J) .IN. SETF)) THEN
      IF (SETF % CARD < MAX_SET_CARD) THEN
        SETF % CARD = SETF % CARD + 1
        SETF % ELEMENT (SETF % CARD) = V (J)
      ELSE
        ! Maximum set size exceeded . . .
      END IF
    END IF
  END DO
END FUNCTION SETF

FUNCTION VECTOR (A) ! Transfer the values of set A into a vector in ascending order
  INTEGER, INTENT (IN) :: A
  INTEGER, POINTER :: VECTOR (:)
  INTEGER I, J, K
  ALLOCATE (VECTOR (A % CARD))
  VECTOR = A % ELEMENT (1 : A % CARD)
  DO I = 1, A % CARD - 1 ! Use a better sort if A % CARD is large
    DO J = I + 1, A % CARD
      IF (VECTOR (I) > VECTOR (J)) THEN
        K = VECTOR (J); VECTOR (J) = VECTOR (I); VECTOR (I) = K
      END IF
    END DO
Examples of using INTEGER_SETS (A, B, and C are variables of type SET; X is an integer variable):

! Check to see if A has more than 10 elements
IF (CARDINALITY (A) > 10) ...

! Check for X an element of A but not of B
IF (X .IN. (A - B)) ...

! C is the union of A and the result of B intersected with the integers 1 to 100
C = A + B * SETF ([[(I, I = 1, 100)]]

! Does A have any even numbers in the range 1:100?
IF (CARDINALITY (A * SETF ([[(I, I = 2, 100, 2)]])) > 0) ...

PRINT *, VECTOR (B) ! Print out the elements of set B, in ascending order

C.9.3.7 Public entities renamed (14.2.2)

At times it might be necessary to rename entities that are accessed with USE statements.

The following example illustrates renaming features of the USE statement.

MODULE J; REAL JX, JY, JZ; END MODULE J

MODULE K
USE J, ONLY : KX => JX, KY => JY
! KX and KY are local names to module K
REAL KZ ! KZ is local name to module K
REAL JZ ! JZ is local name to module K
END MODULE K

PROGRAM RENAME
USE J; USE K
! Module J’s entity JX is accessible under names JX and KX
! Module J’s entity JY is accessible under names JY and KY
! Module K’s entity KZ is accessible under name KZ
! Module J’s entity JZ and K’s entity JZ are different entities
! and cannot be referenced
...
END PROGRAM RENAME

C.9.4 Modules with submodules (14.2.3)

Each submodule specifies that it is the child of exactly one parent module or submodule. Therefore, a module and all of its descendant submodules stand in a tree-like relationship one to another.
A separate module procedure that is declared in a module to have public accessibility can be accessed by use association even if it is defined in a submodule. No other entity in a submodule can be accessed by use association. Each program unit that references a module by use association depends on it, and each submodule depends on its ancestor module. Therefore, if one changes a separate module procedure body in a submodule but does not change its corresponding module procedure interface, a tool for automatic program translation would not need to reprocess program units that reference the module by use association. This is so even if the tool exploits the relative modification times of files as opposed to comparing the result of translating the module to the result of a previous translation.

By constructing taller trees, one can put entities at intermediate levels that are shared by submodules at lower levels; changing these entities cannot change the interpretation of anything that is accessible from the module by use association. Developers of modules that embody large complicated concepts can exploit this possibility to organize components of the concept into submodules, while preserving the privacy of entities that are shared by the submodules and that ought not to be exposed to users of the module. Putting these shared entities at an intermediate level also prevents cascades of reprocessing and testing if some of them are changed.

The following example illustrates a module, color_points, with a submodule, color_points_a, that in turn has a submodule, color_points_b. Public entities declared within color_points can be accessed by use association. The submodules color_points_a and color_points_b can be changed without causing retranslation of program units that reference the module color_points.

The module color_points does not have a module-subprogram-part, but a module-subprogram-part is not prohibited. The module could be published as definitive specification of the interface, without revealing trade secrets contained within color_points_a or color_points_b. Of course, a similar module without the module prefix in the interface bodies would serve equally well as documentation – but the procedures would be external procedures. It would make little difference to the consumer, but the developer would forfeit all of the advantages of modules.

```fortran
module color_points

  type color_point
    private
    real :: x, y
    integer :: color
  end type color_point

  interface ! Interfaces for procedures with separate bodies in the submodule color_points_a
    module subroutine color_point_del ( p ) ! Destroy a color_point object
      type(color_point), allocatable :: p
    end subroutine color_point_del
    ! Distance between two color_point objects
    real module function color_point_dist ( a, b )
      type(color_point), intent(in) :: a, b
    end function color_point_dist
    module subroutine color_point_draw ( p ) ! Draw a color_point object
      type(color_point), intent(in) :: p
    end subroutine color_point_draw
    module subroutine color_point_new ( p ) ! Create a color_point object
  end interface

end module color_points
```
The only entities within color_points_a that can be accessed by use association are the separate module procedures that were declared in color_points. If the procedures are changed but their interfaces are not, the interface from program units that access them by use association is unchanged. If the module and submodule are in separate files, utilities that examine the time of modification of a file would notice that changes in the module could affect the translation of its submodules or of program units that reference the module by use association, but that changes in submodules could not affect the translation of the parent module or program units that reference it by use association.

The variable instance_count in the following example is not accessible by use association of color_points, but is accessible within color_points_a, and its submodules.

```
submodule ( color_points ) color_points_a ! Submodule of color_points

integer :: instance_count = 0

interface
    ! Interface for a procedure with a separate body in submodule color_points_b
    ! palette_stuff, especially submodules thereof, can reference color_points by use association without causing a circular dependence during translation because this use is not in the module. Furthermore, changes in the module palette_stuff do not affect the translation of color_points.
    type(color_point), intent(in) :: pt
    type(palette), intent(out) :: pal
end subroutine inquire_palette

contains ! Invisible bodies for public separate module procedures declared in the module

module subroutine color_point_del ( p )
type(color_point), allocatable :: p
instance_count = instance_count - 1
deallocate ( p )
end subroutine color_point_del

real module function color_point_dist ( a, b ) result ( dist )
type(color_point), intent(in) :: a, b
```
dist = SQRT ( (b%x - a%x)**2 + (b%y - a%y)**2 )
end function color_point_dist

module subroutine color_point_new ( p )
type(color_point), allocatable :: p
instance_count = instance_count + 1
allocate ( p )
end subroutine color_point_new

end submodule color_points_a

8 The subroutine inquire_palette is accessible within color_points_a because its interface is declared therein. It is not, however, accessible by use association, because its interface is not declared in the module, color_points. Since the interface is not declared in the module, changes in the interface cannot affect the translation of program units that reference the module by use association.

module palette_stuff
    type :: palette ; ... ; end type palette
contains
    subroutine test_palette ( p )
        ! Draw a color wheel using procedures from the color_points module
        use color_points ! This does not cause a circular dependency because
        ! the "use palette_stuff" that is logically within
        ! color_points is in the color_points_a submodule.
        type(palette), intent(in) :: p
        ...
    end subroutine test_palette
end module palette_stuff

submodule ( color_points:color_points_a ) color_points_b ! Subsidiary**2 submodule
contains
    ! Invisible body for interface declared in the ancestor module
    module subroutine color_point_draw ( p )
        use palette_stuff, only: palette
        type(color_point), intent(in) :: p
        type(palette) :: MyPalette
        ...; call inquire_palette ( p, MyPalette ); ...
    end subroutine color_point_draw

    ! Invisible body for interface declared in the parent submodule
    module procedure inquire_palette
        ... implementation of inquire_palette
    end procedure inquire_palette

    subroutine private_stuff ! not accessible from color_points_a
        ...
    end subroutine private_stuff
There is a use palette_stuff in color_points_a, and a use color_points in palette_stuff. The use palette_stuff would cause a circular reference if it appeared in color_points. In this case, it does not cause a circular dependence because it is in a submodule. Submodules cannot be referenced by use association, and therefore what would be a circular appearance of use palette_stuff is not accessed.

```fortran
program main
  use color_points
  ! "instance_count" and "inquire_palette" are not accessible here
  ! because they are not declared in the "color_points" module.
  ! "color_points_a" and "color_points_b" cannot be referenced by
  ! use association.
  interface draw
    module procedure color_point_draw
  end interface
  type(color_point) :: C_1, C_2
  real :: RC
  ... call color_point_new (c_1) ! body in color_points_a, interface in color_points
  ... call draw (c_1) ! body in color_points_b, specific interface
  ... ! in color_points, generic interface here.
  ... rc = color_point_dist (c_1, c_2) ! body in color_points_a, interface in color_points
  ... call color_point_del (c_1) ! body in color_points_a, interface in color_points
  ... end program main
```

A multilevel submodule system can be used to package and organize a large and interconnected concept without exposing entities of one subsystem to other subsystems.

Consider a Plasma module from a Tokomak simulator. A plasma simulation requires attention at least to fluid flow, thermodynamics, and electromagnetism. Fluid flow simulation requires simulation of subsonic, supersonic, and hypersonic flow. This problem decomposition can be reflected in the submodule structure of the Plasma module:

<table>
<thead>
<tr>
<th>Plasma module</th>
<th>Flow submodule</th>
<th>Thermal submodule</th>
<th>Electromagnetics submodule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subsonic submodule</td>
<td>Supersonic submodule</td>
<td>Hypersonic submodule</td>
<td></td>
</tr>
</tbody>
</table>

Entities can be shared among the Subsonic, Supersonic, and Hypersonic submodules by putting them within the Flow submodule. One then need not worry about accidental use of these entities by use association or by the Thermal or Electromagnetics submodules, or the development of a dependency of correct operation of those...
subsystems upon the representation of entities of the Flow subsystem as a consequence of maintenance. Since these entities are not accessible by use association, if any of them are changed, the new values cannot be accessed in program units that reference the Plasma module by use association; the answer to the question “where are these entities used” is therefore confined to the set of descendant submodules of the Flow submodule.

C.10 Clause 15 notes

C.10.1 Portability problems with external procedures (15.4.3.5)

There is a potential portability problem in a scoping unit that references an external procedure without explicitly declaring it to have the EXTERNAL attribute (8.5.9). On a different processor, the name of that procedure might be the name of a nonstandard intrinsic procedure and in such a case the processor would interpret those procedure references as references to that intrinsic procedure. (On that processor, the program would also be viewed as not conforming to this document because of the references to the nonstandard intrinsic procedure.) Declaration of the EXTERNAL attribute causes the references to be to the external procedure regardless of the availability of an intrinsic procedure with the same name. Note that declaration of the type of a procedure is not enough to make it external, even if the type is inconsistent with the type of the result of an intrinsic procedure of the same name.

C.10.2 Procedures defined by means other than Fortran (15.6.3)

A processor is not required to provide any means other than Fortran for defining external procedures. Among the means that might be supported are the machine assembly language, other high level languages, the Fortran language extended with nonstandard features, and the Fortran language as supported by another Fortran processor (for example, a previously existing FORTRAN 77 processor). The means other than Fortran for defining external procedures, including any restrictions on the structure for organization of those procedures, are not specified by this document.

A Fortran processor might limit its support of procedures defined by means other than Fortran such that these procedures can affect entities in the Fortran environment only on the same basis as procedures written in Fortran. For example, it might not support the value of a local variable from being changed by a procedure reference unless that variable were one of the arguments to the procedure.

C.10.3 Abstract interfaces and procedure pointer components (15.4, 7.5)

This is an example of a library module providing lists of callbacks that the user can register and invoke.

```fortran
MODULE callback_list_module
  !
  ! Type for users to extend with their own data, if they so desire
  !
  TYPE callback_data
  END TYPE
  !
  ! Abstract interface for the callback procedures
  !
  ABSTRACT INTERFACE
```
SUBROUTINE callback_procedure(data)
  IMPORT callback_data
  CLASS(callback_data),OPTIONAL :: data
END SUBROUTINE

END INTERFACE

! The callback list type.
!
TYPE callback_list
  PRIVATE
  CLASS(callback_record),POINTER :: first => NULL()
END TYPE

! Internal: each callback registration creates one of these
!
TYPE,PRIVATE :: callback_record
  PROCEDURE(callback_procedure),POINTER,NOPASS :: proc
  CLASS(callback_record),POINTER :: next
  CLASS(callback_data),POINTER :: data => NULL();
END TYPE
PRIVATE invoke,forward_invoke
CONTAINS
!
! Register a callback procedure with optional data
!
SUBROUTINE register_callback(list, entry, data)
  TYPE(callback_list),INTENT(INOUT) :: list
  PROCEDURE(callback_procedure) :: entry
  CLASS(callback_data),OPTIONAL :: data
  TYPE(callback_record),POINTER :: new,last
  ALLOCATE(new)
  new%proc => entry
  IF (PRESENT(data)) ALLOCATE(new%data,SOURCE=data)
  new%next => list%first
  list%first => new
END SUBROUTINE

! Internal: Invoke a single callback and destroy its record
!
SUBROUTINE invoke(callback)
  TYPE(callback_record),POINTER :: callback
  IF (ASSOCIATED(callback%data)) THEN
    CALL callback%proc(list%first%data)
    DEALLOCATE(callback%data)
  ELSE
    CALL callback%proc
END IF
DEALLOCATE(callback)
END SUBROUTINE

! Call the procedures in reverse order of registration
!
SUBROUTINE invoke_callback_reverse(list)
    TYPE(callback_list),INTENT(INOUT) :: list
    TYPE(callback_record),POINTER :: next,current
    current => list%first
    NULLIFY(list%first)
    DO WHILE (ASSOCIATED(current))
        next => current%next
        CALL invoke(current)
        current => next
    END DO
END SUBROUTINE

! Internal: Forward mode invocation
!
SUBROUTINE forward_invoke(callback)
    IF (ASSOCIATED(callback%next)) CALL forward_invoke(callback%next)
    CALL invoke(callback)
END SUBROUTINE

! Call the procedures in forward order of registration
!
SUBROUTINE invoke_callback_forward(list)
    TYPE(callback_list),INTENT(INOUT) :: list
    IF (ASSOCIATED(list%first)) CALL forward_invoke(list%first)
END SUBROUTINE

END

C.10.4 Pointers and targets as arguments (15.5.2.4, 15.5.2.6, 15.5.2.7)

1 If a dummy argument is declared to be a pointer, the corresponding actual argument could be a pointer or could be a nonpointer variable or procedure. Consider the two cases separately.

Case (i): The actual argument is a pointer. When procedure execution commences the pointer association status of the dummy argument becomes the same as that of the actual argument. If the pointer association status of the dummy argument is changed, the pointer association status of the actual argument changes in the same way.

Case (ii): The actual argument is not a pointer. This only occurs when the actual argument has the TARGET attribute or is a procedure, and the dummy argument has the INTENT (IN) attribute. The dummy argument becomes pointer associated with the actual argument.

2 When execution of a procedure completes, any pointer that remains defined and that is associated with a dummy argument that has the TARGET attribute and is either a scalar or an assumed-shape array, remains associated
with the corresponding actual argument if the actual argument has the TARGET attribute and is not an array
section with a vector subscript.

For example, consider:

```fortran
REAL, POINTER :: PBEST
REAL, TARGET :: B (10000)
CALL BEST (PBEST, B) ! Upon return PBEST is associated
... ! with the 'best' element of B
CONTAINS
  SUBROUTINE BEST (P, A)
    REAL, POINTER, INTENT (OUT) :: P
    REAL, TARGET, INTENT (IN) :: A (:)
    ... ! Find the 'best' element A(I)
    P => A (I)
  RETURN
  END SUBROUTINE BEST
END
```

When procedure BEST completes, the pointer PBEST is associated with an element of B.

An actual argument without the TARGET attribute can become associated with a dummy argument with the
TARGET attribute. This enables a pointer to become associated with the dummy argument during execution of
the procedure that contains the dummy argument. For example:

```fortran
INTEGER LARGE(100,100)
CALL SUB (LARGE)
... CALL SUB ()
CONTAINS
  SUBROUTINE SUB(ARG)
    INTEGER, TARGET, OPTIONAL :: ARG(100,100)
    INTEGER, POINTER, DIMENSION(:,:) :: PARG
    IF (PRESENT(ARG)) THEN
      PARG => ARG
    ELSE
      ALLOCATE (PARG(100,100))
      PARG = 0
    ENDIF
    ... ! Code with lots of references to PARG
    IF (.NOT. PRESENT(ARG)) DEALLOCATE(PARG)
  END SUBROUTINE SUB
END
```

Within subroutine SUB the pointer PARG is either associated with the dummy argument ARG or it is associated
with an allocated target. The bulk of the code can reference PARG without further calls to the intrinsic function
PRESENT.

If a nonpointer dummy argument has the TARGET attribute and the corresponding actual argument does not,
any pointers that become associated with the dummy argument, and therefore with the actual argument, during
execution of the procedure, become undefined when execution of the procedure completes.

C.10.5 Polymorphic Argument Association (15.5.2.9)

The following example illustrates the polymorphic argument association rules using the derived types defined in Note 7.55.

```fortran
TYPE(POINT) :: T2
TYPE.COLOR_POINT) :: T3
CLASS(POINT) :: P2
CLASS.COLOR_POINT) :: P3

! Dummy argument is polymorphic and actual argument is of fixed type
SUBROUTINE SUB2 ( X2 ); CLASS(POINT) :: X2; ...
SUBROUTINE SUB3 ( X3 ); CLASS.COLOR_POINT) :: X3; ...

CALL SUB2 ( T2 ) ! Valid -- The declared type of T2 is the same as the
! declared type of X2.
CALL SUB2 ( T3 ) ! Valid -- The declared type of T3 is extended from
! the declared type of X2.
CALL SUB3 ( T2 ) ! Invalid -- The declared type of T2 is neither the
! same as nor extended from the declared type
! type of X3.
CALL SUB3 ( T3 ) ! Valid -- The declared type of T3 is the same as the
! declared type of X3.

! Actual argument is polymorphic and dummy argument is of fixed type
SUBROUTINE TUB2 ( D2 ); TYPE(POINT) :: D2; ...
SUBROUTINE TUB3 ( D3 ); TYPE.COLOR_POINT) :: D3; ...

CALL TUB2 ( P2 ) ! Valid -- The declared type of P2 is the same as the
! declared type of D2.
CALL TUB2 ( P3 ) ! Invalid -- The declared type of P3 differs from the
! declared type of D2.
CALL TUB2 ( P3%POINT ) ! Valid alternative to the above
CALL TUB3 ( P2 ) ! Invalid -- The declared type of P2 differs from the
! declared type of D3.
SELECT TYPE ( P2 ) ! Valid conditional alternative to the above
CLASS IS ( COLOR_POINT ) ! Works if the dynamic type of P2 is the same
! as the declared type of D3, or a type
! extended therefrom.
CLASS DEFAULT

! Cannot work if not.
END SELECT
CALL TUB3 ( P3 ) ! Valid -- The declared type of P3 is the same as the
! declared type of D3.
! Both the actual and dummy arguments are of polymorphic type.
CALL SUB2 ( P2 ) ! Valid -- The declared type of P2 is the same as the
! declared type of X2.
```
CALL SUB2 ( P3 ) ! Valid -- The declared type of P3 is extended from
! the declared type of X2.
CALL SUB3 ( P2 ) ! Invalid -- The declared type of P2 is neither the
! same as nor extended from the declared
! type of X3.
SELECT TYPE ( P2 ) ! Valid conditional alternative to the above
CLASS IS ( COLOR_POINT ) ! Works if the dynamic type of P2 is the
CALL SUB3 ( P2 ) ! same as the declared type of X3, or a
! type extended therefrom.
CLASS DEFAULT
! Cannot work if not.
END SELECT
CALL SUB3 ( P3 ) ! Valid -- The declared type of P3 is the same as the
! declared type of X3.

C.10.6 Rules ensuring unambiguous generics (15.4.3.4.5)

1 The rules in 15.4.3.4.5 are intended to ensure

- that it is possible to reference each specific procedure or binding in the generic collection,
- that for any valid generic procedure reference, the determination of the specific procedure referenced is
  unambiguous, and
- that the determination of the specific procedure or binding referenced can be made before execution of the
  program begins (during compilation).

2 Interfaces of specific procedures or bindings are distinguished by fixed properties of their arguments, specifically
type, kind type parameters, rank, and whether the dummy argument has the POINTER or ALLOCATABLE
attribute. A valid reference to one procedure in a generic collection will differ from another because it has an
argument that the other cannot accept, because it is missing an argument that the other requires, or because one
of these fixed properties is different.

3 Although the declared type of a data entity is a fixed property, polymorphic variables allow for a limited degree
of type mismatch between dummy arguments and actual arguments, so the requirement for distinguishing two
dummy arguments is type incompatibility, not merely different types. (This is illustrated in the BAD6 example
later in this note.)

4 That same limited type mismatch means that two dummy arguments that are not type incompatible can be
distinguished on the basis of the values of the kind type parameters they have in common; if one of them has a
kind type parameter that the other does not, that is irrelevant in distinguishing them.

5 Rank is a fixed property, but some forms of array dummy arguments allow rank mismatches when a procedure is
referenced by its specific name. In order to allow rank to always be usable in distinguishing generics, such rank
mismatches are disallowed for those arguments when the procedure is referenced as part of a generic. Additionally,
the fact that elemental procedures can accept array arguments is not taken into account when applying these rules,
so apparent ambiguity between elemental and nonelemental procedures is possible; in such cases, the reference is
interpreted as being to the nonelemental procedure.

6 For procedures referenced as operators or defined-assignment, syntactically distinguished arguments are mapped
to specific positions in the argument list, so the rule for distinguishing such procedures is that it be possible to

distinguish the arguments at one of the argument positions.

For **defined input/output** procedures, only the \texttt{dtv} argument corresponds to something explicitly written in the

program, so it is the \texttt{dtv} that is required to be distinguished. Because \texttt{dtv} arguments are required to be scalar,

ey cannot differ in \texttt{rank}. Thus this rule effectively involves only type and kind type parameters.

For generic procedure names, the rules are more complicated because optional arguments can be omitted and

because arguments can be specified either positionally or by name.

In the special case of type-bound procedures with \texttt{passed-object dummy arguments}, the passed-object argument

is syntactically distinguished in the reference, so rule (3) in 15.4.3.4.5 can be applied. The type of passed-object

arguments is constrained in ways that prevent passed-object arguments in the same \texttt{scoping unit} from being type

incompatible. Thus this rule effectively involves only kind type parameters and \texttt{rank}.

The primary means of distinguishing named generics is rule (4). The most common application of that rule is a

single argument satisfying both (4a) and (4b):

```fortran
INTERFACE GOOD1
  FUNCTION F1A(X)
    REAL :: F1A, X
  END FUNCTION F1A
  FUNCTION F1B(X)
    INTEGER :: F1B, X
  END FUNCTION F1B
END INTERFACE GOOD1
```

Whether one writes \texttt{GOOD1(1.0)} or \texttt{GOOD1(X=1.0)}, the reference is to \texttt{F1A} because \texttt{F1B} would require an integer

argument whereas these references provide the real constant 1.0.

This example and those that follow are expressed using interface bodies, with type as the distinguishing property.

This was done to make it easier to write and describe the examples. The principles being illustrated are equally

applicable when the procedures get their explicit interfaces in some other way or when kind type parameters or

\texttt{rank} are the distinguishing property.

Another common variant is the argument that satisfies (4a) and (4b) by being required in one specific and

completely missing in the other:

```fortran
INTERFACE GOOD2
  FUNCTION F2A(X)
    REAL :: F2A, X
  END FUNCTION F2A
  FUNCTION F2B(X, Y)
    COMPLEX :: F2B
    REAL :: X, Y
  END FUNCTION F2B
END INTERFACE GOOD2
```

Whether one writes \texttt{GOOD2(0.0,1.0)}, \texttt{GOOD2(0.0,Y=1.0)}, or \texttt{GOOD2(Y=1.0,X=0.0)}, the reference is to \texttt{F2B},
because \texttt{F2A} has no argument in the second position or with the name \texttt{Y}. This approach is used as an alternative
to optional arguments when one wants a function to have different result type, kind type parameters, or rank, depending on whether the argument is present. In many of the intrinsic functions, the DIM argument works this way.

It is possible to construct cases where different arguments are used to distinguish positionally and by name:

```fortran
INTERFACE GOOD3
    SUBROUTINE S3A(W,X,Y,Z)
        REAL :: W,Y
        INTEGER :: X,Z
    END SUBROUTINE S3A
    SUBROUTINE S3B(X,W,Z,Y)
        REAL :: W,Z
        INTEGER :: X,Y
    END SUBROUTINE S3B
END INTERFACE GOOD3
```

If one writes `GOOD3(1.0,2,3.0,4)` to reference `S3A`, then the third and fourth arguments are consistent with a reference to `S3B`, but the first and second are not. If one switches to writing the first two arguments as keyword arguments in order for them to be consistent with a reference to `S3B`, the latter two arguments will also need to be written as keyword arguments, `GOOD3(X=2,W=1.0,Z=4,Y=3.0)`, and the named arguments `Y` and `Z` are distinguished.

The ordering requirement in rule (4) is critical:

```fortran
INTERFACE BAD4  ! this interface is invalid!
    SUBROUTINE S4A(W,X,Y,Z)
        REAL :: W,Y
        INTEGER :: X,Z
    END SUBROUTINE S4A
    SUBROUTINE S4B(X,W,Z,Y)
        REAL :: X,Y
        INTEGER :: W,Z
    END SUBROUTINE S4B
END INTERFACE BAD4
```

In this example, the positionally distinguished arguments are `Y` and `Z`, and it is `W` and `X` that are distinguished by name. In this order it is possible to write `BAD4(1.0,2,Y=3.0,Z=4)`, which is a valid reference for both `S4A` and `S4B`.

Rule (1) can be used to distinguish some cases that are not covered by rule (4):

```fortran
INTERFACE GOOD5
    SUBROUTINE S5A(X)
        REAL :: X
    END SUBROUTINE S5A
    SUBROUTINE S5B(Y,X)
        REAL :: Y,X
    END SUBROUTINE S5B
END INTERFACE GOOD5
```
In attempting to apply rule (4), position 2 and name $Y$ are distinguished, but they are in the wrong order, just like the BAD4 example. However, when we try to construct a similarly ambiguous reference, we get GOOD5(1.0, X=2.0), which can’t be a reference to S5A because it would be attempting to associate two different actual arguments with the dummy argument $X$. Rule (4) catches this case by recognizing that S5B requires two real arguments, and S5A cannot possibly accept more than one.

The application of rule (1) becomes more complicated when extensible types are involved. If FRUIT is an extensible type, PEAR and APPLE are extensions of FRUIT, and BOSC is an extension of PEAR, then

```
INTERFACE BAD6  ! this interface is invalid!
  SUBROUTINE S6A(X,Y)
    CLASS(PEAR) :: X,Y
  END SUBROUTINE S6A
  SUBROUTINE S6B(X,Y)
    CLASS(FRUIT) :: X
    CLASS(BOSC) :: Y
  END SUBROUTINE S6B
END INTERFACE BAD6
```

might, at first glance, seem distinguishable this way, but because of the limited type mismatching allowed, BAD6(A_PEAR, A_BOSC) is a valid reference to both S6A and S6B.

It is important to try rule (1) for each type that appears:

```
INTERFACE GOOD7
  SUBROUTINE S7A(X,Y,Z)
    CLASS(PEAR) :: X,Y,Z
  END SUBROUTINE S7A
  SUBROUTINE S7B(X,Z,W)
    CLASS(FRUIT) :: X
    CLASS(BOSC) :: Z
    CLASS(APPLE), OPTIONAL :: W
  END SUBROUTINE S7B
END INTERFACE GOOD7
```

Looking at the most general type, S7A has a minimum and maximum of 3 FRUIT arguments, while S7B has a minimum of 2 and a maximum of three. Looking at the most specific, S7A has a minimum of 0 and a maximum of 3 BOSC arguments, while S7B has a minimum of 1 and a maximum of 2. However, when we look at the intermediate, S7A has a minimum and maximum of 3 PEAR arguments, while S7B has a minimum of 1 and a maximum of 2. Because S7A’s minimum exceeds S7B’s maximum, they can be distinguished.

In identifying the minimum number of arguments with a particular set of properties, we exclude optional arguments and test TKR compatibility, so the corresponding actual arguments are required to have those properties. In identifying the maximum number of arguments with those properties, we include the optional arguments and test not distinguishable, so we include actual arguments which could have those properties but are not required to have them.
These rules are sufficient to ensure that references to procedures that meet them are unambiguous, but there remain examples that fail to meet these rules but which can be shown to be unambiguous:

```fortran
INTERFACE BAD8 ! this interface is invalid !
  ! despite the fact that it is unambiguous !
SUBROUTINE S8A(X,Y,Z)
  REAL,OPTIONAL :: X
  INTEGER :: Y
  REAL :: Z
END SUBROUTINE S8A

SUBROUTINE S8B(X,Z,Y)
  INTEGER,OPTIONAL :: X
  INTEGER :: Z
  REAL :: Y
END SUBROUTINE S8B

END INTERFACE BAD8
```

This interface fails rule (4) because there are no required arguments that can be distinguished from the positionally corresponding argument, but in order for the mismatch of the optional arguments not to be relevant, the later arguments need to be specified as keyword arguments, so distinguishing by name does the trick. This interface is nevertheless invalid so a standard-conforming Fortran processor is not required to do such reasoning. The rules to cover all cases are too complicated to be useful.

The real data objects that would be valid arguments for S9A are entirely disjoint from procedures that are valid arguments to S9B and S9C, and the procedures that valid arguments for S9B are disjoint from the procedures that are valid arguments to S9C because the former are required to accept real arguments and the latter integer arguments. Again, this interface is invalid, so a standard-conforming Fortran processor need not examine such properties when deciding whether a generic collection is valid. Again, the rules to cover all cases are too complicated to be useful.

If one dummy argument has the POINTER attribute and a corresponding argument in the other interface body has the ALLOCATABLE attribute the generic interface is not ambiguous. If one dummy argument has either the POINTER or ALLOCATABLE attribute and a corresponding argument in the other interface body has neither attribute, the generic interface might be ambiguous.

### C.11 Clause 16 notes

#### C.11.1 Atomic memory consistency

##### C.11.1.1 Relaxed memory model

Parallel programs sometimes have apparently impossible behavior because data transfers and other messages can be delayed, reordered and even repeated, by hardware, communication software, and caching and other forms of optimization. Requiring processors to deliver globally consistent behavior is incompatible with performance on many systems. This document specifies that all ordered actions will be consistent (5.3.5 and 11.6), but all consistency between unordered segments is deliberately left processor dependent. Depending on the hardware, this can be observed even when only two images and one mechanism are involved.
C.11.1.2 Examples with atomic operations

1 When variables are being referenced (atomically) from segments that are unordered with respect to the segment that is atomically defining or redefining the variables, the results are processor dependent. This supports use of so-called “relaxed memory model” architectures, which can enable more efficient execution on some hardware implementations.

2 The following examples assume these declarations:

```
MODULE EXAMPLE
  USE, INTRINSIC :: ISO_FORTRAN_ENV
  INTEGER(ATOMICS_INT_KIND) :: X[*] = 0, Y[*] = 0, TMP
```

3 **Example 1**

With X[j] and Y[j] still in their initial state (both zero), image j executes the following sequence of statements:

```
CALL ATOMIC_DEFINE (X, 1)
CALL ATOMIC_DEFINE (Y, 1)
```

and a different image, k, executes the following sequence of statements:

```
DO
  CALL ATOMIC_REF (TMP, Y[j])
  IF (TMP==1) EXIT
END DO
CALL ATOMIC_REF (TMP, X[j])
PRINT *, TMP
```

4 The final value of TMP on image k could be either 0 or 1. That is, even though image j thinks that it defined X[j] before it defined Y[j], this ordering is not guaranteed to be observed on image k. There are many aspects of hardware and software implementation that can cause this effect, but conceptually this example can be thought of as the change in the value of Y propagating faster through the inter-image connections than the change in the value of X.

5 Even if image j executed the sequence

```
CALL ATOMIC_DEFINE (X, 1)
SYNC MEMORY
CALL ATOMIC_DEFINE (Y, 1)
```

the same effect could be seen. That is because even though X and Y are defined in ordered segments, the references from image k are both from a segment that is unordered with respect to image j.

6 Only if the reference on image k to Y[j] is in a segment that is ordered after the segment on image j that defined Y, will TMP be guaranteed to have the value 1.

7 **Example 2:**

With the initial state of X and Y on image j (i.e. X[j] and Y[j]) still being zero, execution of

```
CALL ATOMIC_REF (TMP, X[j])
CALL ATOMIC_DEFINE (Y[j], 1)
PRINT *, TMP
```
on image \( k_1 \), and execution of

\[
\begin{align*}
\text{CALL ATOMIC_REF (TMP, Y} [j]) \\
\text{CALL ATOMIC_DEFINE (X} [j], 1) \\
\text{PRINT *, TMP}
\end{align*}
\]

on image \( k_2 \), in unordered segments, might print the value 1 both times.

This can happen by such mechanisms as “load buffering”; one might imagine that what is happening is that the definitions (ATOMIC_DEFINE) are overtaking the references (ATOMIC_REF). On some processors it is possible that insertion of SYNC MEMORY statements between the calls to ATOMIC_REF and ATOMIC_DEFINE might be sufficient to make the output print the value 1 at most one time (or even exactly one time), but this is still processor dependent unless the SYNC MEMORY statement executions cause the relevant segments on images \( k_1 \) and \( k_2 \) to be ordered.

Example 3:

Because there are no segment boundaries implied by collective subroutines, with the initial state as before, execution of

\[
\begin{align*}
\text{IF (THIS_IMAGE ()==1) THEN} \\
\text{CALL ATOMIC_DEFINE (X} [3], 23) \\
\text{Y = 42} \\
\text{END IF} \\
\text{CALL CO_BROADCAST (Y, 1)} \\
\text{IF (THIS_IMAGE ()==2) THEN} \\
\text{CALL ATOMIC_REF (TMP, X} [3]) \\
\text{PRINT *, Y, TMP} \\
\text{END IF}
\end{align*}
\]

could print the values 42 and 0.

Example 4:

Assuming the declarations

\[
\begin{align*}
\text{INTEGER (ATOMIC_INT_KIND) :: X} [*] = 0, Z = 0
\end{align*}
\]

the statements

\[
\begin{align*}
\text{CALL ATOMIC_ADD (X} [1], 1) \quad \text{! (A)} \\
\text{IF (THIS_IMAGE() == 2) THEN} \\
\text{wait: DO} \\
\text{CALL ATOMIC_REF (Z, X} [1]) \quad \text{! (B)} \\
\text{IF (Z == NUM_IMAGES ()) EXIT wait} \\
\text{END DO wait} \quad \text{! (C)} \\
\text{END IF}
\end{align*}
\]

will execute the “wait” loop on image 2 until all images have completed statement (A). The updates of X [1] are performed by each image in the same manner, but in an arbitrary order. Because the result from the complete set of updates will eventually become visible by execution of statement (B) for some loop iteration on image 2, the termination condition is guaranteed to be eventually fulfilled, provided that no image failure occurs, every
image executes the above code, and no other code is executed in an unordered segment that performs an update
to X [1]. Furthermore, if two SYNC MEMORY statements are inserted in the above code before statement (A)
and after statement (C), respectively, the segment started by the second SYNC MEMORY on image 2 is ordered
after the segments on all images that end with the first SYNC MEMORY.

C.11.2 EVENT_QUERY example

1 The following example illustrates the use of events via a program in which image 1 acts as the master image,
distributing work items to the other images. Only one work item at a time can be active on a worker image, and
each deals with the result (e.g. via input/output) without directly feeding data back to the master image.
2 Because the work items are not expected to be balanced, the master keeps cycling through all images to find one
that is waiting for work.
3 An event is posted by each worker to indicate that it has completed its work item. Since the corresponding
variables are needed only on the master, we place them in an allocatable array component of a coarray. An event
on each worker is needed for the master to post the fact that it has made a work item available for it.

Example code:

PROGRAM work_share
USE, INTRINSIC :: ISO_FORTRAN_ENV, ONLY: EVENT_TYPE
USE :: mod_work, ONLY: & ! Module that creates work items
  work, & ! Type for holding a work item
  create_work_item, & ! Function that creates work item
  process_item, & ! Function that processes an item
  work_done ! Logical function that returns true
            ! if all work has been done.

  TYPE :: worker_type
    TYPE (EVENT_TYPE), ALLOCATABLE :: free (:)
END TYPE

  TYPE (EVENT_TYPE) :: submit [*] ! Post when work ready for a worker
  TYPE (worker_type) :: worker [*] ! Post when worker is free
  TYPE (work) :: work_item [*] ! Holds the data for a work item
  INTEGER :: count, i, nbusy [*]

IF (THIS_IMAGE ()==1) THEN
  ! Get started
  ALLOCATE (worker%free (2:NUM_IMAGES ()))
  nbusy = 0 ! This holds the number of workers working
  DO i = 2, NUM_IMAGES () ! Start the workers working
    IF (work_done ()) EXIT
    nbusy = nbusy + 1
    work_item [i] = create_work_item ()
    EVENT POST (submit [i])
  END DO
  ! Main work distribution loop

  IF (THIS_IMAGE ()==1) THEN
    ! Get started
    ALLOCATE (worker%free (2:NUM_IMAGES ()))
    nbusy = 0 ! This holds the number of workers working
    DO i = 2, NUM_IMAGES () ! Start the workers working
      IF (work_done ()) EXIT
      nbusy = nbusy + 1
      work_item [i] = create_work_item ()
      EVENT POST (submit [i])
    END DO
  END IF

  ! Distribute work items
  DO i = 2, NUM_IMAGES ():
    IF (nbusy /= 0) THEN
      ALLOCATE (worker%free (i+1))
      nbusy = nbusy - 1
      work_item [i] = create_work_item ()
      EVENT POST (submit [i])
    END IF
  END DO
master: DO
  image: DO i = 2, NUM_IMAGES ()
    CALL EVENT_QUERY (worker%free (i), count)
    IF (count==0) CYCLE image ! Worker is not free
    EVENT WAIT (worker%free (i))
    nbusy = nbusy - 1
    IF (work_done ()) CYCLE
    nbusy = nbusy + 1
    work_item [i] = create_work_item ()
    EVENT POST (submit [i])
  END DO image
  IF (nbusy==0) THEN
    ! All done. Exit on all images.
    DO i = 2, NUM_IMAGES ()
      EVENT POST (submit [i])
    END DO
    EXIT master
  END IF
END DO master
ELSE
  ! Work processing loop
  worker: DO
    EVENT WAIT (submit)
    IF (nbusy[1] == 0) EXIT
    CALL process_item (work_item)
    EVENT POST (worker [1]%free (THIS_IMAGE ()))
  END DO worker
END IF
END PROGRAM work_share

C.11.3 Collective subroutine examples

1 The following example computes a dot product of two scalar coarrays using CO_SUM to store the result in a noncoarray scalar variable.

   SUBROUTINE codot (x, y, x_dot_y)
     REAL :: x [*], y [*], x_dot_y
     x_dot_y = x*y
     CALL CO_SUM (x_dot_y)
   END SUBROUTINE codot

2 The function below demonstrates passing a noncoarray dummy argument to CO_MAX. The function uses CO_MAX to find the maximum value of the dummy argument across all images. Then the function flags all images that hold values matching the maximum. The function then returns the maximum image index for an image that holds the maximum value.

   FUNCTION find_max (j) RESULT (j_max_location)

INTEGER, INTENT (IN) :: j
INTEGER j_max, j_max_location
j_max = j
CALL CO_MAX (j_max)
! Flag images that hold the maximum j.
IF (j==j_max) THEN
  j_max_location = THIS_IMAGE()
ELSE
  j_max_location = 0
END IF
! Return highest image index associated with a maximal j.
CALL CO_MAX(j_max_location)
END FUNCTION find_max

C.12 Clause 18 notes

C.12.1 Runtime environments (18.1)

1 This document allows programs to contain procedures defined by means other than Fortran. That raises the
issues of initialization of and interaction between the runtime environments involved.

2 Implementations are free to solve these issues as they see fit, provided that

- heap allocation/deallocation (e.g., (DE)ALLOCATE in a Fortran subprogram and malloc/free in a C func-
  tion) can be performed without interference,
- input/output to and from external files can be performed without interference, as long as procedures defined
  by different means do not do input/output with the same external file,
- input/output preconnections exist as required by the respective standards, and
- initialized data are initialized according to the respective standards.

C.12.2 Example of Fortran calling C (18.3)

C Function Prototype:
int C_Library_Function(void* sendbuf, int sendcount, int *recvcounts);

Fortran Module:

MODULE CLIBFUN_INTERFACE
  INTERFACE
    INTEGER (C_INT) FUNCTION C_LIBRARY_FUNCTION (SENDBUF, SENDCOUNT, RECVCOUNTS) &
      BIND(C, NAME='C_Library_Function')
    USE, INTRINSIC :: ISO_C_BINDING
    IMPLICIT NONE
    TYPE (C_PTR), VALUE :: SENDBUF
    INTEGER (C_INT), VALUE :: SENDCOUNT
    INTEGER (C_INT), VALUE :: RECVCOUNTS(*)
  END FUNCTION C_LIBRARY_FUNCTION
END MODULE CLIBFUN_INTERFACE
END INTERFACE
END MODULE CLIBFUN_INTERFACE

The module CLIBFUN_INTERFACE contains the declaration of the Fortran dummy arguments, which correspond to the C formal parameters. The NAME= is used in the BIND attribute in order to handle the case-sensitive name change between Fortran and C from “c_library_function” to “C_Library_Function”.

The first C formal parameter is the pointer to void sendbuf, which corresponds to the Fortran dummy argument SENDBUF, which has the type _C_PTR and the VALUE attribute.

The second C formal parameter is the int sendcount, which corresponds to the Fortran dummy argument SENDCOUNT, which has the type INTEGER (C_INT) and the VALUE attribute.

The third C formal parameter is the pointer to int recvcounts, which corresponds to the Fortran dummy argument RECVCOUNTS, which is an assumed-size array of type INTEGER (C_INT).

This example show how C_Library_Function might be referenced in a Fortran program unit:

USE, INTRINSIC :: ISO_C_BINDING, ONLY: C_INT, C_FLOAT, C_LOC
USE CLIBFUN_INTERFACE
...
REAL (C_FLOAT), TARGET :: SEND(100)
INTEGER (C_INT) :: SENDCOUNT, RET
INTEGER (C_INT), ALLOCATABLE :: RECVCOUNTS(:)
...
ALLOCATE( RECVCOUNTS(100) )
...
RET = C_LIBRARY_FUNCTION(C_LOC(SEND), SENDCOUNT, RECVCOUNTS)
...

The first Fortran actual argument is a reference to the function C_LOC which returns the value of the C address of its argument, SEND. This value becomes the value of the first formal parameter, the pointer sendbuf, in C_Library_Function.

The second Fortran actual argument is SENDCOUNT of type INTEGER (C_INT). Its value becomes the initial value of the second formal parameter, the int sendcount, in C_Library_Function.

The third Fortran actual argument is the allocatable array RECVCOUNTS of type INTEGER (C_INT). The base C address of this array becomes the value of the third formal parameter, the pointer recvcounts, in C_Library_Function. Note that interoperability is based on the characteristics of the dummy arguments in the specified interface and not on those of the actual arguments. Thus, the fact that the actual argument is allocatable is not relevant here.

C.12.3 Example of C calling Fortran (18.3)

Fortran Code:

SUBROUTINE SIMULATION(ALPHA, BETA, GAMMA, DELTA, ARRAYS) BIND(C)
  USE, INTRINSIC :: ISO_C_BINDING
IMPLICIT NONE

INTEGER (C_LONG), VALUE :: ALPHA
REAL (C_DOUBLE), INTENT(INOUT) :: BETA
INTEGER (C_LONG), INTENT(OUT) :: GAMMA
REAL (C_DOUBLE), DIMENSION(*) , INTENT(IN) :: DELTA

TYPE, BIND(C) :: PASS
  INTEGER (C_INT) :: LENC, LENF
  TYPE (C_PTR) :: C, F
END TYPE PASS

TYPE (PASS), INTENT(INOUT) :: ARRAYS

REAL (C_FLOAT), ALLOCATABLE, TARGET, SAVE :: ETA(:)
REAL (C_FLOAT), POINTER :: C_ARRAY(:)

! Associate C_ARRAY with an array allocated in C
CALL C_F_POINTER (ARRAYS%C, C_ARRAY, [ARRAYS%LENC])

! Allocate an array and make it available in C
ARRAYS%LENF = 100
ALLOCATE (ETA(ARRAYS%LENF))
ARRAYS%F = C_LOC(ETA)

END SUBROUTINE SIMULATION

C Struct Declaration:
struct pass {
  int lenc, lenf;
  float *c, *f;
};

C Function Prototype:
void simulation(long alpha, double *beta, long *gamma, double delta[],
  struct pass *arrays);

C Calling Sequence:
simulation(alpha, beta, gamma, delta, arrays);

1 The above-listed Fortran code specifies a subroutine SIMULATION. This subroutine corresponds to the C void function simulation.

2 The Fortran subroutine references the intrinsic module ISO_C_BINDING.

3 The first Fortran dummy argument of the subroutine is ALPHA, which has the type INTEGER(C_LONG) and the VALUE attribute. This dummy argument corresponds to the C formal parameter alpha, which is a long. The C actual argument is also a long.
The second Fortran dummy argument of the subroutine is BETA, which has the type REAL(C_DOUBLE) and the INTENT (INOUT) attribute. This dummy argument corresponds to the C formal parameter \texttt{beta}, which is a pointer to double. An address is passed as the C actual argument.

The third Fortran dummy argument of the subroutine is GAMMA, which has the type INTEGER(C_LONG) and the INTENT (OUT) attribute. This dummy argument corresponds to the C formal parameter \texttt{gamma}, which is a pointer to long. An address is passed as the C actual argument.

The fourth Fortran dummy argument is the assumed-size array DELTA, which has the type REAL (C_DOUBLE) and the INTENT (IN) attribute. This dummy argument corresponds to the C formal parameter \texttt{delta}, which is a double array. The C actual argument is also a double array.

The fifth Fortran dummy argument is ARRAYS, which is a structure for accessing an array allocated in C and an array allocated in Fortran. The lengths of these arrays are held in the components LENC and LENF; their C addresses are held in components C and F.

C.12.4 Example of calling C functions with noninteroperable data (18.10)

Many Fortran processors support 16-byte real numbers, which might not be supported by the C processor. Assume a Fortran programmer wants to use a C procedure from a message passing library for an array of these reals. The C prototype of this procedure is

```c
void ProcessBuffer(void *buffer, int n_bytes);
```

with the corresponding Fortran interface

```fortran
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 ! The 'C address' of the array buffer
    INTEGER (C_INT), VALUE :: N_BYTES ! Number of bytes in buffer
    END SUBROUTINE PROCESS_BUFFER
END INTERFACE
```

This can be done using \texttt{C_LOC} if the particular Fortran processor specifies that \texttt{C_LOC} returns an appropriate address:

```fortran
REAL(R_QUAD), DIMENSION(:), ALLOCATABLE, TARGET :: QUAD_ARRAY
...
CALL PROCESS_BUFFER(C_LOC(QUAD_ARRAY), INT(16*SIZE(QUAD_ARRAY),C_INT))
! One quad real takes 16 bytes on this processor
```

C.12.5 Example of opaque communication between C and Fortran (18.3)

The following example demonstrates how a Fortran processor can make a modern object-oriented random number generator written in Fortran available to a C program.

```fortran
USE, INTRINSIC :: ISO_C_BINDING
    ! Assume this code is inside a module
```
TYPE RANDOM_STREAM
  ! A (uniform) random number generator (URNG)
CONTAINS
  PROCEDURE(RANDOM_UNIFORM), DEFERRED, PASS(STREAM) :: NEXT
  ! Generates the next number from the stream
END TYPE RANDOM_STREAM

ABSTRACT INTERFACE
  ! Abstract interface of Fortran URNG
  SUBROUTINE RANDOM_UNIFORM(STREAM, NUMBER)
    IMPORT :: RANDOM_STREAM, C_DOUBLE
    CLASS(RANDOM_STREAM), INTENT(INOUT) :: STREAM
    REAL(C_DOUBLE), INTENT(OUT) :: NUMBER
  END SUBROUTINE RANDOM_UNIFORM
END INTERFACE

2 A polymorphic object with declared type RANDOM_STREAM is not interoperable with C. However, we can
make such a random number generator available to C by packaging it inside another nonpolymorphic, nonpara-
meterized derived type:

TYPE :: URNG_STATE ! No BIND(C), as this type is not interoperable
  CLASS(RANDOM_STREAM), ALLOCATABLE :: STREAM
END TYPE URNG_STATE

3 The following two procedures will enable a C program to use our Fortran uniform random number generator:

! Initialize a uniform random number generator:
  SUBROUTINE INITIALIZE_URNG(STATE_HANDLE, METHOD) &
    BIND(C, NAME="InitializeURNG")
    TYPE(C_PTR), INTENT(OUT) :: STATE_HANDLE
    ! An opaque handle for the URNG
    CHARACTER(C_CHAR), DIMENSION(*), INTENT(IN) :: METHOD
    ! The algorithm to be used
    TYPE(URNG_STATE), POINTER :: STATE
    ! An actual URNG object
    ALLOCATE(STATE)
    ! There needs to be a corresponding finalization
    ! procedure to avoid memory leaks, not shown in this example
    ! Allocate STATE%STREAM with a dynamic type depending on METHOD
    ...
    STATE_HANDLE=C_LOC(STATE)
    ! Obtain an opaque handle to return to C
  END SUBROUTINE INITIALIZE_URNG
! Generate a random number:
SUBROUTINE GENERATE_UNIFORM(STATE_HANDLE, NUMBER) &
   BIND(C, NAME="GenerateUniform")
   TYPE(C_PTR), INTENT(IN), VALUE :: STATE_HANDLE
   ! An opaque handle: Obtained via a call to INITIALIZE_URNG
   REAL(C_DOUBLE), INTENT(OUT) :: NUMBER
   TYPE(URNG_STATE), POINTER :: STATE
   ! A pointer to the actual URNG
   CALL C_F_POINTER(CPTR=STATE_HANDLE, FPTR=STATE)
   ! Convert the opaque handle into a usable pointer
   CALL STATE%STREAM%NEXT(NUMBER)
   ! Use the type-bound procedure NEXT to generate NUMBER
END SUBROUTINE GENERATE_UNIFORM

C.12.6 Using assumed type to interoperate with C

C.12.6.1 Overview

The mechanism for handling unlimited polymorphic entities whose dynamic type is interoperable with C is designed to handle the following two situations:

1. A formal parameter that is a C pointer to void. This is an address, and no further information about the entity is provided. The formal parameter corresponds to a dummy argument that is a nonallocatable nonpointer scalar or is an assumed-size array.

2. A formal parameter that is the address of a C descriptor. Additional information on the status, type, size, and shape is implicitly provided. The formal parameter corresponds to a dummy argument that is assumed-shape or assumed-rank.

In the first situation, it is the programmer’s responsibility to explicitly provide any information needed on the status, type, size, and shape of the entity.

C.12.6.2 Mapping of interfaces with void * C parameters to Fortran

A C interface for message passing or input/output functionality could be provided in the form

```c
int EXAMPLE_send(const void *buffer, size_t buffer_size, const HANDLE_t *handle);
```

where the `buffer_size` argument is given in units of bytes, and the `handle` argument (which is of a type aliased to `int`) provides information about the target the buffer is to be transferred to. In this example, type resolution is not required.

The first method provides a thin binding; a call to `EXAMPLE_send` from Fortran directly invokes the C function.

```f90
INTERFACE
   INTEGER (C_INT) FUNCTION example_send(buffer, buffer_size, handle) &
      BIND(C, NAME='EXAMPLE_send')
   USE, INTRINSIC :: ISO_C_BINDING
END INTERFACE
```
It is assumed that this interface is declared in the specification part of the module MOD_EXAMPLE_OLD. An example of its use follows:

```fortran
USE, INTRINSIC :: ISO_C_BINDING
USE MOD_EXAMPLE_OLD

REAL(C_FLOAT) :: x(100)
INTEGER(C_INT) :: y(10,10)
REAL(C_DOUBLE) :: z
INTEGER(C_INT) :: status, handle

! Assign values to x, y, z and initialize handle.

! Send values in x, y, and z using EXAMPLE_send.
status = example_send(x, C_SIZEOF(x), handle)
status = example_send(y, C_SIZEOF(y), handle)
status = example_send(z, C_SIZEOF(z), handle)
```

In those invocations, x and y are passed directly with sequence association, but it is necessary to make an array expression containing the value of z to pass it.

The second method provides a Fortran interface which is easier to use, but requires writing a separate C wrapper routine. With this method, a C descriptor is created because the buffer is assumed-rank in the Fortran interface; the use of an optional argument is also demonstrated.

```fortran
INTERFACE
    SUBROUTINE example_send(buffer, handle, status) BIND(C, NAME="EG_send_fortran")
        USE, INTRINSIC :: ISO_C_BINDING
        TYPE(*), CONTIGUOUS, INTENT (IN) :: buffer(..)
        INTEGER (C_INT), INTENT (IN) :: handle
        INTEGER (C_INT), INTENT(OUT), OPTIONAL :: status
    END SUBROUTINE
END INTERFACE
```

It is assumed that this interface is declared in the specification part of a module MOD_EXAMPLE_NEW. Example invocations from Fortran are then

```fortran
USE, INTRINSIC :: iso_c_binding
USE mod_example_new

TYPE, BIND(C) :: my_derived
```
INTEGER(C_INT) :: len_used
REAL(C_FLOAT) :: stuff(100)

END TYPE

TYPE(my_derived) :: w(3)
REAL(C_FLOAT) :: x(100)
INTEGER(C_INT) :: y(10,10)
REAL(C_DOUBLE) :: z
INTEGER(C_INT) :: status, handle

! Assign values to w, x, y, z and initialize handle.
! Send values in w, x, y, and z using example_send.
CALL example_send(w, handle, status)
CALL example_send(x, handle)
CALL example_send(y, handle)
CALL example_send(z, handle)
CALL example_send(y(:,5), handle) ! Fifth column of y.
CALL example_send(y(1,5), handle) ! Scalar y(1,5) passed by descriptor.

The wrapper routine can be written in C as follows.

#include "ISO_Fortran_binding.h"

void EXAMPLE_send_fortran(const CFI_cdesc_t *buffer, const HANDLE_t *handle, int *status)
{
    int status_local;
    size_t buffer_size;
    int i;

    buffer_size = buffer->elem_len;
    for (i=0; i<buffer->rank; i++) {
        buffer_size *= buffer->dim[i].extent;
    }
    status_local = EXAMPLE_send(buffer->base_addr,buffer_size, handle);
    if (status != NULL) *status = status_local;
}

C.12.7 Using assumed-type variables in Fortran

An assumed-type dummy argument in a Fortran procedure can be used as an actual argument corresponding to an assumed-type dummy in a call to another procedure. In the following example, the Fortran subroutine SIMPLE_SEND serves as a wrapper to hide the complications associated with calls to a C function named ACTUAL_Send. Module COMM_INFO contains node and address information for the current data transfer operations.

SUBROUTINE SIMPLE_SEND(buffer, nbytes)
    USE comm_info, ONLY: my_node, r_node, r_addr
    USE, INTRINSIC :: ISO_C_BINDING

    ! Assign values to w, x, y, z and initialize handle.
    ! Send values in w, x, y, and z using example_send.
    CALL example_send(w, handle, status)
    CALL example_send(x, handle)
    CALL example_send(y, handle)
    CALL example_send(z, handle)
    CALL example_send(y(:,5), handle) ! Fifth column of y.
    CALL example_send(y(1,5), handle) ! Scalar y(1,5) passed by descriptor.

    The wrapper routine can be written in C as follows.

    #include "ISO_Fortran_binding.h"

    void EXAMPLE_send_fortran(const CFI_cdesc_t *buffer, const HANDLE_t *handle, int *status)
    {
        int status_local;
        size_t buffer_size;
        int i;

        buffer_size = buffer->elem_len;
        for (i=0; i<buffer->rank; i++) {
            buffer_size *= buffer->dim[i].extent;
        }
        status_local = EXAMPLE_send(buffer->base_addr,buffer_size, handle);
        if (status != NULL) *status = status_local;
    }

C.12.7 Using assumed-type variables in Fortran
IMPLICIT NONE

TYPE(*), INTENT (IN) :: buffer(*)
INTEGER :: nbytes, ierr

INTERFACE
SUBROUTINE actual_Send(buffer, nbytes, node, addr, ierr) &
BIND(C, NAME="ACTUAL_Send")
IMPORT :: C_SIZE_T, C_INT, C_INTPTR_T
TYPE(*), INTENT (IN) :: buffer(*)
INTEGER(C_SIZE_T), VALUE :: nbytes
INTEGER(C_INT), VALUE :: node
INTEGER(C_INTPTR_T), VALUE :: addr
INTEGER(C_INT), INTENT(OUT) :: ierr
END SUBROUTINE actual_Send
END INTERFACE

CALL actual_Send(buffer, INT(nbytes, C_SIZE_T), r_node, r_addr, ierr)

IF (ierr /= 0) THEN
    PRINT *, "Error sending from node", my_node, "to node", r_node
    PRINT *, "Program Aborting" ! Or call a recovery procedure
    ERROR STOP ! Omit in the recovery case
END IF
END SUBROUTINE simple_Send

C.12.8 Simplifying interfaces for arbitrary rank procedures

1 There are situations where an assumed-rank dummy argument can be useful in Fortran, although a Fortran procedure cannot itself access its value. For example, the IEEE inquiry functions in Clause 14 could be written using an assumed-rank dummy argument instead of writing 16 separate specific routines, one for each possible rank.

2 In particular, the specific procedures for the IEEE_SUPPORT_DIVIDE function could possibly be implemented in Fortran as follows:

INTERFACE ieee_support_divide
    MODULE PROCEDURE ieee_support_divide_noarg, ieee_support_divide_onearg_r, &
    ieee_support_divide_onearg_d
END INTERFACE ieee_support_divide

...)

LOGICAL FUNCTION ieee_support_divide_noarg ()
    ieee_support_divide_noarg = .TRUE.
END FUNCTION ieee_support_divide_noarg
LOGICAL FUNCTION ieee_support_divide_onearg_r (x)
   REAL, INTENT (IN) :: x(..)
   ieee_support_divide_onearg_r4 = .TRUE.
END FUNCTION ieee_support_divide_onearg_r

LOGICAL FUNCTION ieee_support_divide_onearg_d (x)
   DOUBLE PRECISION, INTENT (IN) :: x(..)
   ieee_support_divide_onearg_r8 = .TRUE.
END FUNCTION ieee_support_divide_onearg_d

C.12.9 Processing assumed-shape arrays in C

The example shown below calculates the product of individual elements of arrays A and B and returns the result in array C. The Fortran interface of elemental_mult will accept arguments of any type and rank. However, the C function will return an error code if any argument is not a two-dimensional int array. Note that the arguments are permitted to be array sections, so the C function does not assume that any argument is contiguous.

The Fortran interface is:

INTERFACE
   FUNCTION elemental_mult(a, b, c) BIND(C, NAME="elemental_mult_c") RESULT(err)
      USE, INTRINSIC :: ISO_C_BINDING
      INTEGER(C_INT) :: err
      TYPE(*), DIMENSION(..) :: a, b, c
   END FUNCTION elemental_mult
END INTERFACE

The definition of the C function is:

#include "ISO_Fortran_binding.h"

int elemental_mult_c(CFI_cdesc_t * a_desc, CFI_cdesc_t * b_desc, CFI_cdesc_t * c_desc)
{
   size_t i, j, ni, nj;

   int err = 1; /* this error code represents all errors */

   char * a_col = (char*) a_desc->base_addr;
   char * b_col = (char*) b_desc->base_addr;
   char * c_col = (char*) c_desc->base_addr;
   char *a_elt, *b_elt, *c_elt;

   /* Only support int. */
   if (a_desc->type != CFI_type_int || b_desc->type != CFI_type_int ||
      c_desc->type != CFI_type_int) {
      return err;
   }
/* Only support two dimensions. */
if (a_desc->rank != 2 || b_desc->rank != 2 || c_desc->rank != 2) {
  return err;
}

ni = a_desc->dim[0].extent;
nj = a_desc->dim[1].extent;

/* Ensure the shapes conform. */
if (ni != b_desc->dim[0].extent || ni != c_desc->dim[0].extent) return err;
if (nj != b_desc->dim[1].extent || nj != c_desc->dim[1].extent) return err;

/* Multiply the elements of the two arrays. */
for (j = 0; j < nj; j++) {
  a_elt = a_col;
  b_elt = b_col;
  c_elt = c_col;
  for (i = 0; i < ni; i++) {
    *(int*)a_elt = *)(int*)b_elt * *)(int*)c_elt;
    a_elt += a_desc->dim[0].sm;
    b_elt += b_desc->dim[0].sm;
    c_elt += c_desc->dim[0].sm;
  }
  a_col += a_desc->dim[1].sm;
  b_col += b_desc->dim[1].sm;
  c_col += c_desc->dim[1].sm;
}
return 0;

C.12.10 Creating a contiguous copy of an array

1 A C function might need to create a contiguous copy of an array section, for example, to pass the array section as an actual argument corresponding to a dummy argument with the CONTIGUOUS attribute. The following example provides functions that can be used to copy an array described by a CFI_cdesc_t descriptor to a contiguous buffer. The input array need not be contiguous.

2 The C functions are:

```c
#include "ISO_Fortran_binding.h"
/* Other necessary includes omitted. */

/*
 * Returns the number of elements in the object described by desc.
 * If it is an array, it need not be contiguous.
 * (The number of elements could be zero).
 */
size_t numElements(const CFI_cdesc_t * desc)
```
{  
    CFI_rank_t r;
    size_t num = 1;

    for (r = 0; r < desc->rank; r++) {
        num *= desc->dim[r].extent;
    }

    return num;
}

/*
 * Auxiliary recursive function to copy an array of a given rank.
 * Recursion is useful because an array of rank n is composed of an
 * ordered set of arrays of rank n-1.
 */
static void *copyToContiguous (const CFI_cdesc_t *vald, void *output,
                                const void *input, CFI_rank_t rank)
{
    CFI_index_t e;

    if (rank == 0) {
        /* Copy scalar element. */
        memcpy (output, input, vald->elem_len);
        output = (void *)((char *)output + vald->elem_len);
    }
    else {
        for (e = 0; e < vald->dim[rank-1].extent; e++) {
            /* Recurse on subarrays of lesser rank. */
            output = _copyToContiguous (vald, output, input, rank-1);
            input = (void *) ((char *)input + vald->dim[rank].sm);
        }

        return output;
    }

    /*
    * General routine to copy the elements in the array described by vald
    * to buffer, as done by sequence association. The array itself can
    * be non-contiguous. This is not the most efficient approach.
    */
    void copyToContiguous (void * buffer, const CFI_cdesc_t * vald) {
        _copyToContiguous (vald, buffer, vald->base_addr, vald->rank);
    }
C.12.11 Changing the attributes of an array

A C programmer might want to call more than one Fortran procedure and the attributes of an array involved might differ between the procedures. In this case, it is necessary to set up more than one C descriptor for the array. For example, this code fragment initializes the first C descriptor for an allocatable entity of rank 2, calls a procedure that allocates the array described by the first C descriptor, constructs the second C descriptor by invoking CFI_section with the value CFI_attribute_other for the attribute parameter, then calls a procedure that expects an assumed-shape array.

```c
CFI_CDESC_T(2) loc_alloc, loc_assum;
CFI_cdesc_t * desc_alloc = (CFI_cdesc_t *)&loc_alloc,
   * desc_assum = (CFI_cdesc_t *)&loc_assum;
CFI_index_t extents[2];
CFI_rank_t rank = 2;
int flag;

flag = CFI_establish(desc_alloc,
   NULL,
   CFI_attribute Allocatable,
   CFI_type Double,
   sizeof(double),
   rank,
   NULL);

Fortran_factor (desc_alloc, ...); /* Allocates array described by desc_alloc. */

/* Extract extents from descriptor. */
extents[0] = desc_alloc->dim[0].extent;

flag = CFI_establish(desc_assum,
   desc_alloc->base_addr,
   CFI_attribute Other,
   CFI_type Double,
   sizeof(double),
   rank,
   extents);

Fortran_solve (desc_assum, ...); /* Uses array allocated in Fortran_factor. */
```

After invocation of the second CFI_establish, the lower bounds stored in the dim member of desc_assum will have the value 0 even if the corresponding entries in desc_alloc have different values.

C.12.12 Creating an array section in C using CFI_section

The C function set_odd sets every second element of an array to a specific value, beginning with the first element. If this is done by making an array section descriptor for the elements to be set, and calling a Fortran subroutine
SET_ALL that sets every element of an assumed-shape array to a specific value. An interface block for set_odd permits it to be also called from Fortran.

```fortran
SUBROUTINE set_all(int_array, val) BIND(C)
    INTEGER(C_INT) :: int_array(:)
    INTEGER(C_INT), VALUE :: val
    int_array = val
END SUBROUTINE

INTERFACE
SUBROUTINE set_odd(int_array, val) BIND(C)
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_INT
    INTEGER(C_INT) :: int_array(:)
    INTEGER(C_INT), VALUE :: val
END SUBROUTINE
END INTERFACE

#include "ISO_Fortran_binding.h"

void set_odd(CFI_cdesc_t *int_array, int val)
{
    CFI_index_t lower_bound[1], upper_bound[1], stride[1];
    CFI_CDESC_T(1) array;
    int status;
    /* Create a new descriptor which will contain the section. */
    status = CFI_establish((CFI_cdesc_t *)&array,
                           NULL,
                           CFI_attribute_other,
                           int_array->type,
                           int_array->elem_len,
                           /* rank */ 1,
                           /* extents is ignored */NULL);

    lower_bound[0] = int_array->dim[0].lower_bound;
    upper_bound[0] = lower_bound[0] + (int_array->dim[0].extent - 1);
    stride[0] = 2;

    status = CFI_section((CFI_cdesc_t *)&array,
                           int_array,
                           lower_bound,
                           upper_bound,
                           stride);

    set_all( (CFI_cdesc_t *) &array, val);

    /* Here one could make use of int_array and access all its data. */
}
```
The `set_odd` procedure can be called from Fortran as follows:

```fortran
INTEGER(C_INT) :: d(5)
d = (/ 1, 2, 3, 4, 5 /)
CALL set_odd(d, -1)
PRINT *, d
```

This program will print something like:

```
-1 2 -1 4 -1
```

During execution of the subroutine `SET_ALL`, its dummy argument `INT_ARRAY` would have size (and upper bound) 3.

It is also possible to invoke `set_odd()` from C. However, it would be the C programmer’s responsibility to make sure that all members of the C descriptor have the correct value on entry to the function. Inserting additional checking into the function could alleviate this problem.

Following is an example C function that dynamically generates a C descriptor for an assumed-shape array and calls `set_odd`.

```c
#include <stdio.h>
#include <stdlib.h>
#include "ISO_Fortran_binding.h"

#define ARRAY_SIZE 5

void example_of_calling_set_odd(void)
{
    CFI_CDESC_T(1) d;
    CFI_index_t extent[1];
    CFI_index_t subscripts[1];
    void *base;
    int i, status;

    base = malloc(ARRAY_SIZE*sizeof(int));
    extent[0] = ARRAY_SIZE;
    status = CFI_establish((CFI_cdesc_t *)&d,
                              base,
                              CFI_attribute_other,
                              CFI_type_int,
                              /* element length is ignored */ 0,
                              /* rank */ 1,
                              extent);

    set_odd((CFI_cdesc_t *)&d, -1);

    for (i=0; i<ARRAY_SIZE; i++) {
        subscripts[0] = i;
    }
```
printf(" %d",*((int *)CFI_address((CFI_cdesc_t *)&d, subscripts)));
}
putc(10, stdout);
free(base);
}

The above C function will print similar output to that of the preceding Fortran program.

C.12.13 Use of CFI_setpointer

1 The C function change_target modifies a pointer to an integer variable to become associated with a global variable defined inside C:

```c
#include "ISO_Fortran_binding.h"

int y = 2;

void change_target(CFI_cdesc_t *ip) {
    CFI_CDESC_T(0) yp;
    int status;
    /* Make local yp point at y. */
    status = CFI_establish((CFI_cdesc_t *)&yp,
        &y,
        CFI_attribute_pointer,
        CFI_type_int,
        /* elem_len is ignored */ sizeof(int),
        /* rank */ 0,
        /* extents are ignored */ NULL);
    /* Pointer-associate ip with (the target of) yp. */
    status = CFI_setpointer(ip, (CFI_cdesc_t *)&yp, NULL);
    if (status != CFI_SUCCESS) {
        ...
        report run time error...
    }
}
```

2 The restrictions on the use of CFI_establish prohibit direct modification of the incoming pointer entity ip by invoking that function on it.

3 The following program illustrates the usage of change_target from Fortran.

```fortran
PROGRAM change_target_example
    USE, INTRINSIC :: ISO_C_BINDING
    INTERFACE
        SUBROUTINE change_target(ip) BIND(C)
            IMPORT :: C_INT
            INTEGER(C_INT), POINTER :: ip
        END SUBROUTINE
    END INTERFACE
```
INTEGER(C_INT), TARGET :: it = 1
INTEGER(C_INT), POINTER :: it_ptr
it_ptr => it
WRITE (*,*) it_ptr
CALL change_target(it_ptr)
WRITE (*,*) it_ptr

4 This will print something similar to

1
2

C.12.14 Mapping of MPI interfaces to Fortran

1 The Message Passing Interface (MPI) specifies procedures for exchanging data between MPI processes. This example shows the usage of MPI_Send and is similar to the second variant of EXAMPLE_Send in C.12.6.2. It also shows the usage of assumed-length character dummy arguments and optional dummy arguments.

2 MPI_Send has the C prototype:

    int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag,
                 MPI_Comm comm);

where MPI_Datatype and MPI_Comm are opaque handles. Most MPI C functions return an error code, which in Fortran is the last dummy argument to the corresponding subroutine and can be made optional. Thus, the use of a Fortran subroutine requires a wrapper function, declared as

    void MPI_Send_f(CFI_cdesc_t *buf, int count, MPI_Datatype_f datatype, int dest, int tag, MPI_Datatype_f comm, int *ierror);

This wrapper function will convert MPI_Datatype_f and MPI_Comm_f to MPI_Datatype and MPI_Comm, and produce a contiguous void * buffer from CFI_cdesc_t *buf (if necessary).

4 Similarly, the wrapper function for MPI_Comm_set_name could have the C prototype:

    void MPI_Comm_set_name_f(MPI_Comm comm, CFI_cdesc_t *comm_name, int *ierror);

5 The Fortran handle types and interfaces are defined in the module MPI_F08. For example,

    MODULE mpi_f08
    ...
    TYPE, BIND(C) :: mpi_comm
    PRIVATE
    INTEGER(C_INT) :: mpi_val
    END TYPE mpi_comm

    INTERFACE
    SUBROUTINE MPI_SEND(buf,count,datatype,dest,tag,comm,ierror) &
    BIND(C, NAME='MPI_Send_f')
    USE, INTRINSIC :: ISO_C_BINDING
IMPORT :: MPI_Datatype, MPI_Comm
TYPE(*), DIMENSION(..), INTENT (IN) :: buf
INTEGER(C_INT), VALUE, INTENT (IN) :: count, dest, tag
TYPE(mpi_datatype), INTENT (IN) :: datatype
TYPE(mpi_comm), INTENT (IN) :: comm
INTEGER(C_INT), OPTIONAL, INTENT (OUT) :: ierror
END SUBROUTINE mpi_send

SUBROUTINE mpi_comm_set_name(comm,comm_name,ierror) &
BIND(C, NAME='MPI_Comm_set_name_f')
USE, INTRINSIC :: ISO_C_BINDING
IMPORT :: mpi_comm
TYPE(mpi_comm), INTENT (IN) :: comm
CHARACTER(KIND=C_CHAR, LEN=*) :: comm_name
INTEGER(C_INT), OPTIONAL, INTENT (OUT) :: ierror
END SUBROUTINE mpi_comm_set_name
END INTERFACE

END MODULE mpi_f08

6 Some examples of invocation from Fortran are:

USE, INTRINSIC :: ISO_C_BINDING
USE :: MPI_f08

TYPE(mpi_comm) :: comm
REAL :: x(100)
INTEGER :: y(10,10)
REAL(KIND(1.0d0)) :: z
INTEGER :: dest, tag, ierror

! Assign values to x, y, z and initialize MPI variables.

! Set the name of the communicator.
CALL mpi_comm_set_name(comm, "Communicator Name", ierror)

! Send values in x, y, and z.
CALL mpi_send(x, 100, MPI_REAL, dest, tag, comm, ierror)
IF (ierror/=0) PRINT *, 'WARNING: X send error', ierror
CALL mpi_send(y(3,:), 10, MPI_INTEGER, dest, tag, comm)
CALL mpi_send(z, 1, MPI_DOUBLE_PRECISION, dest, tag, comm)

7 The first example sends the entire array X and includes the optional error argument return value. The second example sends a noncontiguous subarray (the third row of Y) and the third example sends a scalar Z. Note the differences between the calls in this example and those in C.12.6.2.
C.13 Clause 19 notes : Examples of host association (19.5.1.4)

The first two examples are examples of valid host association. The third example is an example of invalid host association.

Example 1:

```fortran
PROGRAM A
  INTEGER I, J
  ... CONTAINS
    SUBROUTINE B
      INTEGER I ! Declaration of I hides
      ! program A's declaration of I
      ... I = J ! Use of variable J from program A
      ! through host association
    END SUBROUTINE B
END PROGRAM A
```

Example 2:

```fortran
PROGRAM A
  TYPE T
  ... CONTAINS
    SUBROUTINE B
      IMPLICIT TYPE (T) (C) ! Refers to type T declared below
      ! in subroutine B, not type T declared above in program A
      ... TYPE T
      ... END TYPE T
    END SUBROUTINE B
END PROGRAM A
```

Example 3:

```fortran
PROGRAM Q
  REAL (KIND = 1) :: C
  ... CONTAINS
    SUBROUTINE R
      REAL (KIND = KIND (C)) :: D ! Invalid declaration
    END SUBROUTINE R
END PROGRAM Q
```
In the declaration of D in subroutine R, the use of C would refer to the declaration of C in subroutine R, not program Q. However, it is invalid because the declaration of C is required to occur before it is used in the declaration of D (10.1.12).
Index

In the index, entries in italic denote BNF terms, and page numbers in bold face denote primary text or definitions.

Symbols

- 157
<, 161, 450
<=, 161
>, 161
>=, 161
**, 157
+, 157
-stmt, 20
.EQ., 151, 153, 156, 160, 161, 161, 163, 304, 450
.EQV., 152, 153, 156, 160, 160
.FALSE., 68, 483
.GE., 151, 153, 156, 160, 161, 163, 304, 450
.GT., 151, 153, 156, 160, 161, 163, 304, 450
.LE., 151, 153, 156, 160, 161, 163, 304, 450
.LT., 151, 153, 156, 160, 161, 163, 304, 450
.NE., 151, 153, 156, 160, 161, 161, 163, 304, 450
.NEQV., 152, 153, 156, 160, 160, 417
.NOT., 152, 153, 156, 160, 160
.OR., 152, 153, 156, 160, 160, 357
.TRUE., 68, 483
/, 157
/ edit descriptor, 283
//, 159
//=, 161, 450
: edit descriptor, 283
; 54
<=, 450
==, 161, 450
>, 450
>=, 450
& 54, 290

A

A edit descriptor, 280
ABS, 354, 452
ABSTRACT, 69, 70, 85, 304, 305
ABSTRACT attribute, 23, 69, 85
abstract interface, 15, 15, 296, 303, 305, 311, 330, 351, 518
abstract interface block, 15, 15, 305
abstract type, 23, 59, 82, 85, 85, 88, 133, 141
ac-do-variable (R776), 93, 93, 94, 166, 168, 516
ac-implied-do (R774), 93, 93, 94, 155, 516
ac-implied-do-control (R775), 93, 93, 155, 166–168, 516
ac-spec (R770), 93, 93
ac-value (R773), 93, 93, 94
access-id (R828), 113, 113
access-name, 113
access-spec (R807), 69, 74, 75, 80–83, 97, 100, 100, 113, 114, 306, 311
access-stmt (R827), 36, 100, 113, 113, 114
ACCESS= specifier, 228, 230, 256, 257
accessibility attribute, 100, 113, 296
accessibility statement, 113
ACHAR, 68, 172, 354
ACOS, 354
ACOSH, 354
ACQUIRED_LOCK= specifier, 215, 530, 533
action, 220
action-stmt (R515), 7, 37, 37, 155, 198, 206
ACTION= specifier, 228, 230, 256, 257, 567
active image, 14, 142, 143, 146, 147, 188, 214, 217, 345, 411
assignment, 169–182
  defined, 81, 173, 308
elemental, 12, 173, 174
elemental array (FORALL), 180
masked array (WHERE), 177
pointer, 174
assignment statement, 16, 17, 44, 58, 84, 169, 181, 479, 528
assignment-stmt (R1032), 37, 169, 169, 178, 180, 181, 532
ASSOCIATE construct, 184, 187, 323, 516, 517, 520, 532
associate name, 6, 6, 23, 58, 61, 88, 145, 184, 185, 188, 204, 517, 520, 521, 526, 532
ASSOCIATE statement, 184, 520
associate-construct (R1102), 37, 184, 184
associate-construct-name, 184
associate-stmt (R1103), 7, 184, 184, 206
ASSOCIATED, 75, 144, 147, 342
association, 6
  argument, 6, 6, 23, 58, 66, 76, 77, 102, 105, 112, 113, 145, 146, 301, 315–317, 326, 333, 517, 523, 526, 527, 542, 588
common, 128
construct, 6, 6, 145, 146, 517, 520, 523, 526
equivalence, 126
host, 6, 6, 39, 59, 60, 66, 100, 113, 116, 121, 128, 166, 167, 175, 299, 301, 323, 335, 337, 516–520, 523, 526, 527, 616
inheritance, 6, 6, 9, 47, 86, 88, 523, 526
linkage, 6, 6, 509, 517, 520, 520
name, 6, 6, 47, 517, 523
sequence, 322
storage, 6, 6, 46, 47, 125–127, 334, 337, 432, 523–526
use, 6, 6, 39, 47, 60, 66, 86, 100, 111, 113, 121, 125–127, 166, 167, 175, 296, 295–299, 305, 333, 337, 516–518, 521
association (R1104), 184, 184
assumed type parameter, 23, 24, 58, 317, 320
assumed-implied-spec (R821), 106, 106, 107
assumed-rank dummy data object, 6, 45, 60, 84, 103, 104, 139, 201, 202, 302, 303, 309, 317–320, 324, 330, 397, 398, 421, 428, 430, 437, 438, 483, 493, 495, 603, 604, 606
assumed-rank-spec (R825), 104, 107
assumed-shape array, 5, 60, 103, 105, 124, 139, 303, 317–319, 321, 324, 330, 483, 493, 495, 586, 603, 610, 612
assumed-shape-spec (R819), 104, 105, 105
assumed-size-spec (R822), 104, 106, 106
assumed-type, 7, 59, 60, 317, 330, 493, 605
ASYNCHRONOUS attribute, 100, 100, 101, 114, 185, 194, 237, 296, 298, 302, 303, 319, 371, 508, 511, 518, 519
asynchronous communication, 100, 511
asynchronous input/output, 100, 228, 230, 232, 236–238, 241, 242, 249, 252, 253, 255, 257, 259, 260
ASYNCHRONOUS statement, 114, 186, 300, 516, 519
ASYNCHRONOUS= specifier, 229, 230, 234, 235, 236, 256, 257
ATAN, 359
ATAN2, 31, 359
ATANH, 360
atomic subroutine, 22, 41, 208, 209, 341, 344, 346, 360–364, 381, 440, 444, 532
ATOMIC_ADD, 360, 441
ATOMIC_AND, 360
ATOMIC_CAS, 361
ATOMIC_DEFINE, 361, 594, 595
ATOMIC_FETCH_ADD, 361
ATOMIC_FETCH_AND, 362
ATOMIC_FETCH_OR, 362
ATOMIC_FETCH_XOR, 363
ATOMIC_INT_KIND, 360–364, 440
ATOMIC_LOGICAL_KIND, 361, 364, 440
ATOMIC_OR, 363
ATOMIC_REF, 363, 594, 595
ATOMIC_XOR, 364
C_NULL_FUNPTR, 483, 484
C_NULL_PTR, 483, 484
C_SHORT, 483
C_SIGNED_CHAR, 483
C_SIZE_T, 483
C_SIZEOF, 107, 167, 488
C_VERTICAL_TAB, 484
CALL statement, 22, 206, 208, 301, 313, 326, 334, 411
call-stmt (R1521), 37, 313, 314, 316
CASE statement, 198
case-construct (R1140), 37, 198, 198, 199
case-construct-name, 198
case-expr (R1144), 198, 198, 199
case-selector (R1145), 198, 198
case-stmt (R1142), 198, 198
case-value (R1147), 199, 199
case-value-range (R1146), 199, 199
CEILING, 368
CFI_address, 500
CFI_allocate, 501, 507
CFI_cdesc_t, 8, 493–495, 496, 496–498, 500–506, 508
CFI_deallocate, 498, 502, 507
CFI_establish, 502
CFI_is_contiguous, 504
CFI_section, 504
CFI_select_part, 505
CFI_setpointer, 506
CHANGE TEAM construct, 22, 46, 187, 193, 205, 516, 517, 532
CHANGE TEAM statement, 22, 40, 46, 187, 208, 217, 345, 520
change-team-construct (R1111), 37, 187, 187
change-team-stmt (R1112), 187, 187
changeable mode, 226
CHAR, 67, 368
char-length (R723), 65, 65, 66, 74, 75, 97–99, 544
char-literal-constant (R724), 51, 55, 56, 66, 248, 269, 270, 535
char-selector (R721), 61, 65, 66
char-string-edit-desc (R1321), 268, 270
char-variable (R905), 131, 131, 225
character context, 8, 49, 53–55, 67
character literal constant, 66
character sequence type, 19, 71, 126–128, 525, 526, 529
character set, 49
character storage unit, 21, 21, 106, 126, 128, 440, 524, 529, 530
character string edit descriptor, 268, 285
character type, 65–68
CHARACTER_KINDS, 440
CHARACTER_STORAGE_SIZE, 440
characteristics, 8, 87, 176, 245, 246, 302, 303, 312, 313, 322, 326, 330, 331, 334, 353, 414
dummy argument, 302
procedure, 302
child data transfer statement, 224, 225, 236–238, 241, 247, 245–249, 265, 288
CLASS, 59, 59, 60, 245
CLASS DEFAULT statement, 203
CLASS IS statement, 203
CLOSE statement, 220, 221, 225, 227, 228, 232, 232, 249, 252, 566
close-spec (R1209), 233, 233
close-stmt (R1208), 37, 233, 337
CMPLX, 171, 172, 343, 369, 449
CO_BROADCAST, 369
CO_MAX, 370
CO_MIN, 370
CO_REDUCE, 371, 539
CO_SUM, 372, 539
coarray-association (R1113), 46, 187, 187
coarray-name, 115, 187, 188, 516, 518
coarray-spec (R809), 74–76, 97, 98, 101, 101, 102, 114, 115, 119
cobound, 8, 45, 46, 101–103, 139, 142, 167, 185, 321, 349, 351, 399, 411, 438, 517
codimension, 8, 8, 10, 45, 103, 139, 185, 302, 373, 399, 438
CODIMENSION attribute, 60, 76, 98, 101, 101, 107, 115
codimension-decl (R835), 115, 115, 185, 187, 188, 516
codimension-stmt (R834), 37, 115, 518
coindexed object, 8, 45, 46, 115, 133, 139, 170, 172, 174, 175, 184, 209, 213, 313, 314, 317–320, 324, 337,
360–364, 381, 411, 485, 487, 488
coindexed-named-object (R914), 131, 132, 134, 134
collating sequence, 8, 67, 68, 161, 272, 354, 368, 369, 390, 393, 400–402, 405–410, 413, 536
collective subroutine, 22, 341, 345, 346, 369–372, 384, 432, 444
COMMAND_ARGUMENT_COUNT, 168, 372, 387
comment, 54, 55, 292
common association, 128
common block storage sequence, 127
COMMON statement, 8, 127, 127–129, 186, 298, 300, 515, 525, 526, 542
common-block-name, 114, 119, 127, 186, 298
common-block-object (R874), 127, 298, 518
common-stmt (R873), 37, 127, 518
companion processor, 7, 9, 15, 42, 47, 48, 69, 91, 101, 483, 488, 509–511, 535
compatibility
Fortran 77, 32
Fortran 2003, 31
Fortran 2008, 30
Fortran 95, 31
COMPILER_OPTIONS, 167, 440
COMPILER_VERSION, 167, 441
completion step, 42, 232
complex part designator, 11, 43, 134
complex type, 64–65
complex-literal-constant (R718), 51, 64
complex-part-designator (R915), 131, 134, 134, 135, 139
component, 9, 10, 14, 16, 19–21, 69–71, 74, 89, 121, 515
direct, 9, 9, 69, 78, 319, 448, 490
parent, 6, 9, 80, 84, 86, 89, 526, 554
potential subobject, 9, 69, 70, 337
component definition statement, 74
component keyword, 16, 47, 80, 89, 515
component order, 9, 80, 89, 240
component-array-spec (R740), 74, 74–76
component-attr-spec (R738), 74, 74, 75, 77, 78
component-data-source (R758), 88, 88–90
component-decl (R739), 66, 74, 74–76, 78
component-def-stmt (R736), 9, 74, 74, 75
component-initialization (R743), 74, 78, 78
collating sequence, 8, 67, 68, 161, 272, 354, 368, 369, 390, 393, 400–402, 405–410, 413, 536
collating sequence, 8, 67, 68, 161, 272, 354, 368, 369, 390, 393, 400–402, 405–410, 413, 536
collating sequence, 8, 67, 68, 161, 272, 354, 368, 369, 390, 393, 400–402, 405–410, 413, 536
collating sequence, 8, 67, 68, 161, 272, 354, 368, 369, 390, 393, 400–402, 405–410, 413, 536
collating sequence, 8, 67, 68, 161, 272, 354, 368, 369, 390, 393, 400–402, 405–410, 413, 536
FOR ALL, 180, 337, 338, 516, 517, 520, 523, 526
IF, 41, 183, 197, 455, 541
nonblock DO, xviii, 542
SELECT CASE, 41, 183, 197, 455, 541
SELECT RANK, xviii, 41, 106, 183, 185, 201, 323, 532
SELECT TYPE, 41, 58, 59, 183, 185, 203, 323, 516, 517, 520, 532
WHERE, 16, 177
construct association, 6, 6, 145, 146, 517, 520, 523, 526
construct entity, 6, 10, 113, 184, 186, 187, 194, 203, 513, 514, 516, 523
construct-name, 205
constructor
array, 93
derived-type, 88
structure, 88
CONTAINS statement, 39, 40, 81, 335
contains-stmt (R1543), 36, 81, 296, 335
contiguous, 10, 20, 30, 71, 78, 103, 132, 139, 176, 177, 185, 194, 195, 237, 244, 397, 488, 524, 545
CONTIGUOUS attribute, 75, 78, 103, 104, 115, 138, 139, 176, 194, 302, 318–321, 324, 493, 495, 525
CONTIGUOUS statement, 115
contiguous-stmt (R836), 37, 115
continuation, 54, 55
CONTINUE statement, 206, 541
continue-stmt (R1159), 37, 191, 206
control character, 49, 67, 219, 222
control edit descriptor, 268, 282–285
control information list, 234
control mask, 178
control-edit-desc (R1313), 268, 269
conversion
numeric, 171
corank, 10, 45, 46, 76, 101, 102, 104, 133, 139, 141, 149, 185, 302, 320, 373, 394, 399, 411, 435, 438, 517
COS, 373
COSH, 373
COSHAPE, 373
cosubscript, 10, 45, 46, 103, 139, 349, 351, 394, 435, 438
cosubscript (R925), 133, 139, 139
COUNT, 342, 374
CPU_TIME, 374
CRITICAL construct, 189, 193, 205
CRITICAL statement, 163, 189, 208, 218
critical-construct (R1116), 37, 189, 189
critical-construct-name, 189
critical-stmt (R1117), 7, 189, 189, 206
CSHIFT, 375
current record, 223
current team, 22, 84, 140, 142, 146, 147, 187, 188, 210, 213–215, 217, 345, 369, 384, 389, 394, 411, 415, 431, 432, 434, 435, 438, 441, 444
CURRENT_TEAM, 388, 389, 441
cycle-stmt (R1133), 37, 187, 193, 193

D
d (R1310), 269, 269, 274–278, 281, 282, 288
D edit descriptor, 275
data edit descriptor, 268, 272–282
data entity, 8, 9, 10, 17–19, 24, 43–45
data object, 7–9, 10, 10, 11, 19, 21, 24, 38–40, 43–46
data object designator, 11, 19, 45, 131
data object reference, 19, 44, 46
data pointer, 17, 17, 46, 485, 497, 525
DATA statement, 32, 40, 93, 99, 115, 128, 300, 415, 516, 519, 527, 542, 543
data transfer, 243
data transfer input statement, 233
data transfer output statement, 233
data type, 22, see type
data-component-def-stmt (R737), 74, 74–76
data-edit-desc (R1307), 268, 269
data-i-do-object (R841), 115, 115, 116
data-i-do-variable (R842), 115, 115, 116, 168, 516
data-implied-do (R840), 115, 115–117, 168, 516
data-pointer-component-name, 174
data-pointer-initialization compatible, 78
data-pointer-object (R1034), 174, 174, 175, 181, 358, 533
data-ref (R911), 7, 132, 133–135, 174, 175, 237, 313, 316, 323, 329
data-stmt (R837), 36, 115, 305, 336, 518
data-stmt-constant (R845), 93, 116, 116, 117
data-stmt-object (R839), 115, 115–117
data-stmt-repeat (R844), 116, 116, 117
data-stmt-set (R88), 115, 115
data-stmt-value (R843), 115, 116, 116
data-target (R1037), 88–90, 111, 174, 174, 175, 181, 323, 337, 358, 524
DATE_AND_TIME, 376
DBLE, 343, 377
DC edit descriptor, 285
dealloc-opt (R942), 144, 145, 145, 146
DEALLOCATE statement, 144, 146, 147, 208, 442, 502, 536
defallocate-stmt (R941), 37, 144, 533
decimal edit descriptor, 285
decimal edit mode, 230
decimal symbol, 10, 230, 237, 257, 272–278, 285, 286
decimal-edit-desc (R1320), 269, 270
DECIMAL= specifier, 229, 230, 234, 237, 249, 256, 257, 285
declarable, 10, 11, 22, 169, 172, 173, 178, 181, 301, 308, 309, 313, 318, 337, 338
defined assignment statement, 31, 173, 326, 532
defined operation, 11, 19, 153, 162, 163–166, 191, 301, 307, 313, 326, 337
defined-to-generic-spec (R1509), 11, 82, 245, 246, 251, 304, 304, 307, 309
defined-operator (R609), 52, 82, 298, 304, 545
defined-unary-op (R1003), 17, 52, 152, 152, 153, 162, 297
definition, 10, 11
definition of variables, 527
deleted features, 29, 30, 32, 33, 541, 542
DELIM= specifier, 229, 230, 234, 235, 237, 249, 256, 258, 291, 293, 567
delimiter mode, 230
derived type, 11, 21, 23, 42, 43, 46, 57, 68–90, 93, 490, 491
derived type definition statement, see TYPE statement
derived type determination, 71
derived-type-type-specifier, 59
derived-type-def (R726), 36, 60, 69, 70, 73, 491
derived-type-spec (R754), 24, 59, 60, 66, 88, 88, 89, 203, 245, 515
derived-type-stmt (R727), 69, 69, 70, 73, 100, 518
descendant, 11, 39, 70, 71, 80, 81, 83, 111, 299, 514
data object, 131
designator (R901), 78, 116, 131, 131, 133, 134, 149, 174, 184, 289, 336, 337
designator, 149
digit, 27, 49, 49, 52, 62, 92, 287
digit-string (R711), 62, 62, 63, 273, 274, 280
digit-string, 62
DIGITS, 377
DIM, 377
DIMENSION attribute, 76, 98, 104, 104, 117, 127
dimension-spec (R814), 104
dimension-stmt (R848), 37, 117, 518
direct access, 221
direct access data transfer statement, 238
direct component, 9, 9, 69, 78, 319, 448, 490
DIRECT= specifier, 256, 258
dummy data object, 6, 8, 12, 60, 77, 99, 106–108, 112, 302, 307–309
assumed-rank, 6, 45, 60, 64, 103, 104, 139, 201, 202, 302, 303, 309, 317–320, 324, 330, 397, 398, 421, 428, 430, 437, 438, 483, 493, 495, 603, 604, 606
dummy function, 12, 66, 98
dummy procedure, 8, 12, 14, 18, 107, 120, 123, 124, 167, 175, 301, 302, 304–306, 310, 311, 314, 322, 328–330, 333, 336, 510, 514, 519
dummy-arg (R1536), 332, 332–334
dummy-arg-name (R1531), 118, 120, 301, 331, 331, 332, 335, 518
dynamic type, 17, 23, 24, 60, 61, 84, 85, 87, 90, 94, 113, 142, 144, 146, 162, 164, 170, 172, 173, 175, 176, 185, 203, 204, 208, 212, 251, 313, 319, 320, 329, 351, 383, 407, 411, 424, 432, 494, 517, 521, 527, 554, 603
distinguishable, 309
DO CONCURRENT construct, 30, 190, 193, 196, 205, 337, 338, 516, 517, 523, 529, 531, 536, 544
DO CONCURRENT statement, 59, 180, 181, 190
DO construct, 41, 52, 94, 190, 205, 240, 541, 556, 557
DO statement, 190, 528, 542, 544
DO WHILE statement, 190
do-construct (R1119), 37, 190, 191, 193, 205
do-construct-name, 190, 191, 193
do-stmt (R1120), 7, 190, 190, 191, 206, 532
do-variable (R1124), 93, 115, 190, 190, 192, 239, 263, 265, 287, 528, 530, 532, 566
DOT_PRODUCT, 377
DOUBLE PRECISION, 53, 61, 63, 69
DP edit descriptor, 285
DPROD, 378
DSHIFTL, 378
DSHIFTR, 379
DT edit descriptor, 282
dte-type-spec (R1221), 245
dummy data object, 6, 8, 12, 60, 77, 99, 106–108, 112, 302, 307–309
assumed-rank, 6, 45, 60, 64, 103, 104, 139, 201, 202, 302, 303, 309, 317–320, 324, 330, 397, 398, 421, 428, 430, 437, 438, 483, 493, 495, 603, 604, 606
dummy function, 12, 66, 98
dummy procedure, 8, 12, 14, 18, 107, 120, 123, 124, 167, 175, 301, 302, 304–306, 310, 311, 314, 322, 328–330, 333, 336, 510, 514, 519
dummy-arg (R1536), 332, 332–334
dummy-arg-name (R1531), 118, 120, 301, 331, 331, 332, 335, 518
dynamic type, 17, 23, 24, 60, 61, 84, 85, 87, 90, 94, 113, 142, 144, 146, 162, 164, 170, 172, 173, 175, 176, 185, 203, 204, 208, 212, 251, 313, 319, 320, 329, 351, 383, 407, 411, 424, 432, 494, 517, 521, 527, 554, 603
E
e (R1311), 269, 269, 275–278, 281, 282, 288
E edit descriptor, 275
edit descriptor, 268
/., 283
;:, 283
A, 280
B, 279
BN, 284
BZ, 284
character string, 268, 285
data, 268, 272–282
dC, 285
decimal, 285
dP, 285
dT, 282
e, 275
EM, 276
ES, 277
EX, 278
F, 274
G, 280, 281
H, 541
I, 273
L, 280
O, 279
P, 284
position, 282
RC, 284
RD, 284
RN, 284
round, 284
RP, 284
RU, 284
RZ, 284
S, 284
SP, 284
SS, 284
T, 283
TL, 283
TR, 283
X, 283
Z, 279
effective argument, 5, 6, 12, 24, 58, 60, 61, 66, 103–108, 209, 316–318, 421, 486, 494, 517, 523, 526, 528, 531
effective item, 12, 240, 243, 244, 248, 249, 251, 263, 270, 271, 283, 286, 287, 291, 538
effective position, 310
element sequence, 322
ELEMENTAL, 13, 329, 330, 334, 336, 338
elemental, 12, 22, 45, 66, 84, 87, 162, 163, 168, 173, 174, 176, 178, 179, 301–303, 311, 312, 318, 322, 326, 328, 334, 338, 341, 346, 365, 366, 412, 454, 455
elemental array assignment (FORALL), 180
elemental assignment, 12, 173, 174
elemental operation, 12, 155, 166, 179
elemental operator, 12, 155, 448
elemental procedure, 12, 45, 166, 175, 311, 314, 323, 327, 330, 338, 338, 341, 342
elemental reference, 13, 179, 318, 326–329, 339
elemental subprogram, 13, 329, 330, 338
ELSE IF statement, 53, 197
ELSE statement, 197
else-if-stmt (R1136), 197, 197
else-stmt (R1137), 197, 197
ELSEWHERE statement, 53, 178
elsewhere-stmt (R1048), 177, 178, 178
EN edit descriptor, 276
ENCODING= specifier, 229, 230, 256, 258, 537
END ASSOCIATE statement, 53, 184
END BLOCK DATA statement, 53, 300
END BLOCK statement, 53, 146, 186
END CRITICAL statement, 53, 163, 189, 208
END DO statement, 53, 191
END ENUM statement, 53, 91
END FORALL statement, 53, 180
END FUNCTION statement, 53, 331
END IF statement, 53, 197, 541
END INTERFACE statement, 53, 304
END MODULE statement, 53, 296
END PROCEDURE statement, 53, 333
END PROGRAM statement, 53, 295
END SELECT statement, 53, 198, 203
END statement, 13, 40, 40, 41, 53, 55, 85, 111, 128, 145, 146, 208, 486, 532
END SUBMODULE statement, 53, 299
END SUBROUTINE statement, 53, 332
END TEAM statement, 40, 187, 208, 217, 345
END TYPE statement, 53, 70
END WHERE statement, 53, 178
do-associate-stmt (R1106), 7, 184, 184, 206
do-block-data-stmt (R1422), 13, 36, 40, 299, 300, 300
do-block-stmt (R1110), 7, 186, 186, 206
do-change-team-stmt (R1114), 187, 187, 188
do-critical-stmt (R1118), 7, 189, 189, 206
do-do (R1131), 190, 191, 191, 193
do-do-stmt (R1132), 7, 191, 191, 206
do-enum-stmt (R763), 91, 91
do forall-stmt (R1054), 180, 180
do-function-stmt (R1533), 7, 13, 20, 35, 40, 206, 304, 331, 331, 335
do-if-stmt (R1138), 7, 197, 197, 206
do-interface-stmt (R1504), 304, 304
do-module-stmt (R1406), 13, 35, 40, 296, 296
do mp-subprogram-stmt (R1540), 7, 13, 20, 36, 40, 206, 333, 333, 335
do-program-stmt (R1403), 7, 13, 20, 35, 40, 42, 85, 206, 207, 295, 295
do-select-rank-stmt (R1151), 7, 201, 201, 206
do-select-stmt (R1143), 7, 198, 198, 199, 206
do-select-type-stmt (R1155), 7, 203, 203, 204, 206
do-submodule-stmt (R1419), 13, 35, 40, 299, 299
do-subroutine-stmt (R1537), 7, 13, 20, 35, 40, 206, 304, 332, 332, 335
do-type-stmt (R730), 69, 70
do-where-stmt (R1049), 178, 178
END= specifier, 7, 234, 235, 252, 253, 263
definition record, 220
ENDFILE statement, 53, 220, 221, 223, 230, 249, 252, 254, 566
defile-stmt (R1225), 37, 253, 337
definition (R803), 66, 75, 97, 98, 99, 167, 168, 518
definition-name, 114, 119
ENTRY statement, 12, 40, 162, 173, 296, 301, 305, 329, 331, 333, 515, 525, 526, 542, 544
definition-name, 331, 333, 334, 515
definition-stmt (R1541), 36, 296, 299, 305, 333, 515, 518
ENUM statement, 91
definition (R759), 36, 91, 92
definition-def-stmt (R760), 91, 91
derived-enum (R762), 91
derived-enum-def-stmt (R761), 91
defined equivalent, 91
defined equivalence, 125
defined equivalence object, 125
defined equivalence set, 125
defined equivalence-stmt (R760), 37, 125, 518
ERF, 380
ERFC, 380
ERFC_SCALED, 381
ERR= specifier, 7, 229, 233, 234, 252, 253, 255–257, 262
ermsg-variable (R930), 140, 140, 142, 145–147, 209, 210, 213, 215, 217, 218, 532, 536
ERRMSG= specifier, xix, 140, 142, 144, 145, 147, 189, 209, 217, 529, 536
derived error indicator, 500
ERROR STOP statement, 41, 42, 206, 536
derived error termination, 42, 85, 143, 145, 206, 248, 263, 345, 360–364, 381, 382, 411, 422, 535, 538
derived error-stop-stmt (R1161), 37, 85, 206
ERROR_UNIT, 226, 227, 231, 441
ES edit descriptor, 277
estimation coarray, 8, 139, 140, 188
evaluation operations, 155
optional, 163
parentheses, 164
EVENT POST statement, 208, 213, 214, 217, 381, 441, 442, 530, 532, 537
event variable, 24, 41, 208, 213, 214, 217, 381, 441, 442, 530
EVENT WAIT statement, 208, 213, 214, 217, 441, 442, 530, 532, 537
event-post-stmt (R1161), 37, 213
event-variable (R1171), 213, 213, 217, 442, 532
event-wait-spec (R1173), 213, 213
EVENT_WAIT-stmt (R1172), 37, 213, 213
EVENT_QUERY, xx, 381, 562, 597
EXECUTE COMMAND_LINE, 351, 382
execution control, 183
executable construct, 20, 20, 39
executable-construct (R514), 20, 36, 37, 333
execute, 20, 20
existence, 220, 227
EXEC= specifier, 256, 258
EXIT statement, 184, 194, 205
exit-stmt (R1156), 37, 187, 205, 205
EXP, 382
explicit formatting, 267–285
explicit initialization, 13, 78, 79, 98, 99, 115, 521, 526, 527
explicit interface, 15, 31, 77, 81, 124, 176, 302–306, 311, 312, 314, 316, 322, 335, 336, 514, 515, 532, 575
explicit-coshape-spec (R811), 101, 102, 102
explicit-shape array, 5, 60, 76, 102, 105, 169, 318, 323, 492
explicit-shape-spec (R816), 5, 74, 75, 99, 104, 104–106, 127
EXPRESSION, 382, 461
exponent (R717), 63, 63
exponent-letter (R716), 63, 63
expr (R1022), 28, 84, 88, 89, 93, 140, 149, 152, 152, 164, 165, 168–175, 179, 181, 184, 198, 239, 313, 314, 335, 337, 479, 532
evalulation expression, 149, 149–169
format (R1215), 233–235, 236, 236, 243, 267, 268
format control, 270
format descriptor, see edit descriptor
FORMAT statement, 29, 40, 52, 236, 267, 267, 296
format-item (R1304), 268, 268
format-items (R1303), 267, 268, 268
format-specification (R1302), 267, 267
FORMAT statement, 36, 267, 267, 296, 299, 305
FORMATTED, 245, 246, 304
formatted data transfer, 244
formatted input/output statement, 219, 235
formatted record, 219
FORMATTED= specifier, 256, 258
formatting
explicit, 267–285
list-directed, 245, 285–289
namelist, 245, 289–294
forms, 220
Fortran 2003 compatibility, 31
Fortran 2008 compatibility, 30
Fortran 77 compatibility, 32
Fortran 90 compatibility, 32
Fortran 95 compatibility, 31
Fortran character set, 49, 65
FRACTION, 385
free source form, 53, 53
function, 13
intrinsic, 341
intrinsic elemental, 341
intrinsic inquiry, 341
function reference, 19, 43, 44, 266
function result, 13, 31, 66, 97, 121, 126, 127, 145, 302, 331, 334, 338, 493, 515, 525, 531
FUNCTION statement, 12, 59, 60, 121, 162, 166, 295, 329, 330, 333, 334, 515
function-name, 98, 305, 331, 334, 335, 515, 518
function-reference (R1520), 89, 98, 131, 149, 313, 316, 326
function-stmt (R1530), 35, 304, 305, 330, 331, 331, 515, 518
function-subprogram (R1529), 21, 35, 36, 296, 331, 333
generic interface block, 15, 15, 305, 306
generic procedure reference, 309
GENERIC statement, 81, 82, 306, 306, 327
generic-name, 82, 83, 304, 515, 518
genric-spec (R1508), 15, 82, 83, 86, 113, 162, 173, 297, 298, 304, 304–306, 327, 515, 518
genric-stmt (R1510), 36, 306
GET_COMMAND, 386
GET_COMMAND_ARGUMENT, 387
GET_ENVIRONMENT_VARIABLE, 388, 539
GET_TEAM, 388
global entity, 513
global identifier, 513
GO TO statement, 7, 53, 206, 206
goto-stmt (R1157), 37, 206, 206
graphic character, 49, 67, 292
H
halting mode, 447, 451, 452, 455, 457, 469, 475, 476, 511, 540
hex-constant (R767), 92, 92
hex-digit (R768), 92, 92, 280
hex-digit-string (R1322), 280, 280
host, 14, 14, 38, 299, 336, 515, 518, 519
host association, 6, 6, 17, 39, 59, 60, 66, 100, 113, 116, 121, 128, 166, 167, 175, 299, 301, 323, 335, 337, 516–520, 523, 526, 616
host instance, 14, 176, 314, 315, 322, 333, 358, 518, 523, 526, 532
host scoping unit, 14, 38, 121, 123, 327, 328, 519, 526
HUGE, 389
HYPOT, 390
I
I edit descriptor, 273
IACHAR, 68, 172, 390
IALL, 390
IAND, 391
IANY, 391
IBCLR, 392
IBITS, 392
IBSET, 392
ICHAR, 67, 68, 393
id-variable (R1214), 234, 234

632 ISO/IEC JTC 1/SC 22/WG5/N2118
IEEE infinity, 14
IEEE NaN, 14, 162, 449, 450, 471
IEEE_ALL, 448
IEEE_ARITHMETIC, 166–168, 353, 447–479, 545
IEEE_AWAY, 451, 458
IEEE_CLASS, 456, 456
IEEE_CLASS_TYPE, 448, 456, 479
IEEE_COPY_SIGN, 453, 456
IEEE_DATATYPE, 448
IEEE_DENORMAL, 448
IEEE_DIVIDE, 448
IEEE_DIVIDE_BY_ZERO, 448
IEEE_DOWN, 448, 451
IEEE_EXCEPTIONS, 166–168, 194, 447–479, 545
IEEE_FEATURES, 447–448
IEEE_FEATURES_TYPE, 448
IEEE_FLAG_TYPE, 448, 457, 469, 475
IEEE_FMA, 457
IEEE_GET_FLAG, 194, 450, 457, 480, 481, 540
IEEE_GET_HALTING_MODE, 194, 457, 458
IEEE_GET_MODES, 452, 458, 458, 469, 470
IEEE_GET_ROUNDING_MODE, 451, 458, 458, 470
IEEE_GET_STATUS, 450, 459, 450, 470, 471, 480, 540
IEEE_GET_UNDERFLOW_MODE, 459, 471
IEEE_HALTING, 448
IEEE_INEXACT, 448
IEEE_INEXACT_FLAG, 448
IEEE_INVALID, 448
IEEE_INT, 459, 540
IEEE_INVALID_FLAG, 448
IEEE_IS_FINITE, 460
IEEE_IS_NEGATIVE, 460
IEEE_IS_NEGATIVE, 460
IEEE_IS_NORMAL, 461
IEEE_LOGB, 453, 461
IEEE_MAX_NUM, 461, 540
IEEE_MAX_NUM_MAG, 462, 540
IEEE_MIN_NUM, 462, 540
IEEE_MIN_NUM_MAG, 463, 540
IEEE_MODES_TYPE, 448, 452, 458, 469, 470
IEEE_NAN, 448
IEEE_NEAREST, 448, 451
IEEE_NEGATIVE_DENORMAL, 448
IEEE_NEGATIVE_INF, 448
IEEE_NEGATIVE_NORMAL, 448
IEEE_NEGATIVE_SUBNORMAL, 448, 448, 456, 460, 461
IEEE_NEGATIVE_ZERO, 448
IEEE_NEXT_AFTER, 453, 463
IEEE_NEXT_DOWN, 464, 464
IEEE_NEXT_UP, 464
IEEE_OTHER, 448
IEEE_OTHER_VALUE, 448
IEEE_OVERFLOW, 448
IEEE_POSITIVE_DENORMAL, 448
IEEE_POSITIVE_INF, 448
IEEE_POSITIVE_NORMAL, 448
IEEE_POSITIVE_SUBNORMAL, 448, 448, 456, 460
IEEE_POSITIVE_ZERO, 448
IEEE_REAL, 467
IEEE_RINT, 453, 467
IEEE_ROUND_TYPE, 448, 459, 467, 470, 477
IEEE_ROUNDING, 448
IEEE_SCALB, 453, 468
IEEE_SELECTED_REAL_KIND, 468
IEEE_SET_FLAG, 450, 459, 469, 470, 480, 481, 540
IEEE_SET_HALTING_MODE, 194, 450, 458, 469, 475, 480, 481, 540
IEEE_SET_MODES, 452, 458, 458, 469, 470
IEEE_SET_ROUNDING_MODE, 451, 458, 458, 470
IEEE_SET_STATUS, 450, 459, 450, 470, 471, 480, 540
IEEE_SET_UNDERFLOW_MODE, 458, 459, 470, 471
IEEE_SIGNALING_EQ, 471
IEEE_SIGNALING_GE, 472
IEEE_SIGNALING_GT, 472
IEEE_SIGNALING_LE, 472
IEEE_SIGNALING_LT, 473

ISO/IEC JTC 1/SC 22/WG5/N2118 633
IEEE_SIGNALING_NAN, 448
IEEE_SIGNALING_NE, 473
IEEE_SIGNBIT, 473
IEEE_SQRT, 448
IEEE_STATUS_TYPE, 448, 452, 459, 470, 471, 480
IEEE_SUBNORMAL, 448
IEEE_SUPPORT_DATATYPE, 448, 449, 456, 457, 459, 461–467, 470–473, 474, 474, 478, 479
IEEE_SUPPORT_DENORMAL, 474
IEEE_SUPPORT_DIVIDE, 475, 478
IEEE_SUPPORT_FLAG, 475, 478
IEEE_SUPPORT_HALTING, 475, 478
IEEE_SUPPORT_INF, 452, 453, 464, 474, 478, 479
IEEE_SUPPORT_IO, 476
IEEE_SUPPORT_NAN, 449, 450, 452, 476, 478, 479
IEEE_SUPPORT_RND, 477, 478
IEEE_SUPPORT_STANDARD, 477
IEEE_SUPPORT_SUBNORMAL, 452, 453, 456, 464, 474, 478, 478, 479
IEEE_SUPPORT_UNDERFLOW_CONTROL, 478
IEEE_TO_ZERO, 448, 451
IEEE_UNDERFLOW, 448
IEEE_UNDERFLOW_FLAG, 448
IEEE_UNORDERED, 453, 479
IEEE_UP, 448, 451
IEEE_USUAL, 448
IEEE_VALUE, 462, 463, 479
IEOR, 364, 393
IF construct, 41, 197, 541
IF statement, 155, 198
if-construct (R1134), 37, 197, 197
if-construct-name, 197
if-stmt (R1139), 37, 198, 198
if-then-stmt (R1135), 7, 197, 197, 206
imag-part (R720), 64, 64
active, 14, 147
failed, 14, 147, 215
stopped, 14
image control statement, 14, 41, 163, 189, 194, 207, 208, 208, 209, 212, 217, 218, 326, 337, 345, 346, 384, 432, 444
image index, 14, 40, 45, 139, 140, 210, 214, 220, 321, 349, 351, 394, 435, 438, 513
image-selector (R924), 8, 10, 40, 133, 134, 139, 139, 140, 289
image-selector-spec (R926), 139, 139
image-set (R1167), 210, 210
IMAGE_INDEX, 394
IMAGE_STATUS, 394
imaginary part, 64
implicit interface, 15, 75, 176, 296, 311–314, 322, 487, 519
IMPLICIT NONE statement, 120
IMPLICIT statement, 40, 120, 125, 300
implicit-none-spec (R866), 120, 120
implicit-part (R505), 36, 36, 186
implicit-part-stmt (R506), 36, 36
implicit-spec (R864), 120, 120
implicit-stmt (R863), 36, 120
implied-shape array, 106
implied-shape-or-assumed-size-spec (R823), 104, 106, 106, 107
implied-shape-spec (R824), 104, 107, 107
IMPORT statement, 40, 123, 319
import-name, 123
import-stmt (R867), 36, 123, 186
IMPURE, 329, 330, 334, 336, 338
IN, 108
INCLUDE line, 53, 55
inclusive scope, 14, 121, 187, 206, 229, 233, 234, 236, 252, 253, 255, 257, 314, 334, 513, 514
INDEX, 394
index-name, 180–182, 190–193, 516, 517, 529
inherit, 6, 9, 14, 70, 82, 84, 86, 87, 526, 554
inheritance association, 6, 6, 9, 47, 86, 88, 523, 526
initial team, 22, 46, 139, 226, 351, 389, 394, 415, 434, 442
initial-data-target (R744), 78, 78, 98, 99, 116, 117
initial-proc-target (R1518), 78, 311, 311, 312
INITIAL_TEAM, 388, 389, 442
initialization, 99
default, 10, 10, 76–79, 88–90, 99, 106, 107, 115, 126–128, 319, 521, 526, 531
explicit, 13, 78, 79, 98, 99, 115, 521, 526, 527
initialization (R805), 98, 98, 99, 168
INOUT, 53, 108

input statement, 233

input-item (R1216), 233, 234, 239, 239, 239, 251, 265, 532

input/output editing, 267–294

input/output list, 238

input/output statement, 528

input/output statements, 219–264

input/output unit, 16, 24, 40

INPUT_UNIT, 226, 227, 231, 247, 442

INQUIRE statement, 32, 221, 222, 224, 225, 227, 228, 237, 238, 248, 249, 252, 255, 264, 265, 443, 528, 530–532, 538, 565

inquire-spec (R1231), 256, 256, 257, 265

inquire-stmt (R1230), 37, 256, 337


inquiry, type parameter, 134

instance, 332

INT, 117, 171, 172, 343, 378, 379, 391, 393, 395, 396, 407

int-constant (R607), 51, 51, 116

int-constant-expr (R1031), 61, 65, 73, 91, 115, 168, 168, 201, 202

int-constant-name, 62

int-constant-subobject (R846), 116, 116

int-expr (R1026), 40, 58, 93, 132, 135, 136, 139, 140, 155, 164, 164, 166–168, 190–192, 206, 216, 213, 214, 225, 226, 229, 234, 239, 241, 252, 256, 334

int-literal-constant (R708), 51, 62, 62, 65, 268, 269

int-variable (R907), 131, 131, 140, 229, 233–235, 252, 253, 255, 256, 259–264, 538

int-variable-name, 190

INT16, 442

INT32, 442

INT64, 442

INT8, 442

integer constant, 62

integer editing, 273

integer model, 344

integer type, 61–62

integer-type-spec (R705), 59, 61, 61, 73, 93, 115, 190, 516

INTEGER_KINDS, 442


INTENT statement, 107, 107–109, 118, 194, 545

INTENT statement, 118, 186

int-spec (R826), 97, 108, 118, 311

int-stmt (R849), 37, 118


abstract, 15, 15, 296, 303, 305, 311, 330, 514, 518

explicit, 15, 31, 77, 81, 124, 176, 302–306, 311, 312, 314, 316, 322, 335, 336, 514, 515, 532, 575


implicit, 15, 75, 176, 296, 311–314, 322, 487, 519

procedure, 303

specific, 15, 251, 305, 305, 306, 311, 327

interface block, 15, 39, 245, 251, 297, 304–306, 327, 575, 576

interface body, 15, 19, 40, 103, 105, 107, 120, 166, 304, 304, 330, 333, 335, 495, 515, 518, 575

INTERFACE statement, 304, 575

interface-block (R1501), 36, 304, 304

interface-body (R1505), 304, 304, 305

interface-name (R1516), 81–83, 310, 311, 311

interface-specification (R1502), 304, 304, 305

interface-stmt (R1503), 304, 304–307, 518

internal file, 16, 16, 219, 225, 226, 228, 236, 240, 243, 245, 247, 249, 262, 264, 282, 283, 528, 530, 538

internal procedure, 14, 18, 38, 175, 301–304, 314, 322, 328, 330, 333, 358, 510, 514, 515, 519

internal subprogram, 22, 38, 40, 121, 301, 327, 518

internal unit, 16, 16, 226, 228, 242, 248, 257, 265, 443

internal-file-variable (R1203), 225, 225, 226, 235, 265,
IPARITY, 396
IS_CONTIGUOUS, 60, 397
IS_IOSTAT_END, 397
IS_IOSTAT_EOR, 398
ISHFT, 396
ISHFTC, 397
ISO 10646 character, 16, 65, 68, 169, 225, 230, 240, 272, 286, 413, 426
ISO_C_BINDING, 60, 75, 85, 101, 107, 141, 142, 166, 167, 172, 337, 440–490, 530–532, 545
K
k (R1314), 269, 269, 276, 281, 284
keyword, 16
  argument, 12, 16, 47, 303, 306, 315, 341, 346, 453, 514, 515, 516, 575
  component, 16, 47, 80, 89, 515
  statement, 16, 47
  type parameter, 16, 47, 88
keyword (R516), 47, 47, 88, 313, 314
KIND, 61–65, 68, 73, 92, 135, 171, 172, 398
kind-param (R709), 62, 62, 63, 65–68
kind-selector (R706), 28, 61, 61, 62, 68
L
L edit descriptor, 280
label, see statement label
  label (R611), 7, 52, 52, 190, 191, 206, 229, 233, 234, 236, 252, 253, 255–257, 263, 264, 314
  label-do-stmt (R1121), 190, 190, 191
language-binding-spec (R808), 97, 101, 114, 115, 330
LBOUND, 60, 170, 176, 185, 202, 318, 398
bracket (R771), 74, 93, 93, 97, 98, 114, 115, 119, 139, 140
LCOBOUND, 399
LEADZ, 399
left tab limit, 283
LEN, 135, 399
LEN_TRIM, 400

length type parameter, 23, 24, 24, 42, 58, 68, 77, 93, 110, 143, 170, 317, 399, 488, 489, 545

length-selector (R722), 28, 65, 65, 66

letter, 49, 49, 50, 52, 120, 150, 152

letter-spec (R865), 120, 120

level-1-expr (R1002), 150, 150, 153

level-2-expr (R1006), 150, 150, 151, 153, 154

level-3-expr (R1010), 151, 151

level-4-expr (R1012), 151, 151, 152

level-5-expr (R1017), 152, 152

lexical token, 14, 16, 27, 50, 52

LGE, 68, 400

LGT, 68, 401

line, 16, 53–56

linkage association, 6, 6, 509, 517, 520, 520

list-directed formatting, 245, 285–289

list-directed input/output statement, 236

literal constant, 9, 44, 132, 164

literal-constant (R605), 51, 51, 149

LLE, 68, 401

LLT, 68, 401

LOCAL, 191, 194, 196, 516, 523, 531

local identifier, 513, 514

local procedure pointer, 17, 332

local variable, 17, 25, 30, 31, 43, 44, 46, 99, 100, 103, 105, 111, 113, 144, 145, 314, 332, 336

local-defined-operator (R1414), 297, 297, 298

local-name, 297, 298

LOCAL_INIT, 191, 194, 196, 516, 523, 529, 531

locality, 194, 195, 516, 523, 529, 531

locality-spec (R1130), 191, 191

LOCK statement, 208, 215, 217, 218, 443–445, 530, 532

lock variable, 25, 41, 217, 218, 443–445, 530, 532

lock-stat (R1179), 215, 215

lock-stmt (R1178), 37, 215

lock-variable (R1181), 215, 215, 217, 443, 532

LOCK_TYPE, 25, 70, 108, 141, 142, 215, 443

LOG, 31, 402

LOG10, 402

LOG_GAMMA, 402

LOGICAL, 403

logical intrinsic operation, 159

logical type, 68

logical-literal-constant (R725), 51, 68, 150, 152

logical-variable (R904), 131, 131, 215, 217, 256, 258, 259, 533

LOGICAL_KINDS, 443

loop-control (R1123), 190, 190, 192, 193, 195

lower-bound (R817), 104, 104–107

lower-bound-expr (R935), 140, 140, 141, 174

lower-cobound (R812), 102, 103, 103

M

m (R1309), 269, 269, 273, 274, 279, 280

main program, 16, 18, 21, 22, 38, 41, 43

main-program (R1401), 35, 38, 123, 295, 295

mask-expr (R1046), 177, 178, 178–181, 190, 191, 193

masked array assignment, 16, 177, 528

masked array assignment (WHERE), 177

masked-elsewhere-stmt (R1047), 177, 178, 178, 181

MASKL, 403

MASKR, 403

MATMUL, 404

MAX, 339, 342, 404

MAXEXPONENT, 405

MAXLOC, 342, 405

MAXVAL, 406

MERGE, 407

MERGE_BITS, 407

MIN, 408

MINEXPONENT, 408

MINLOC, 408

MINVAL, 409

MOD, 32, 410

model

mode

blank interpretation, 230

changeable, 226

collection, 226

decimal edit, 230

delimiter, 230

halting, 447, 451, 452, 455, 457, 469, 475, 476, 511, 540

IEEE rounding, 447, 448, 450, 452

input/output rounding, 226, 232, 238, 261, 279, 284, 476

pad, 231

sign, 232, 284

underflow, 451, 452, 454, 459, 471, 478, 479, 540

model

bit, 343

ISO/IEC JTC 1/SC 22/WG5/N2118 637
extended real, 344
integer, 344
real, 344

module, 16, 17, 18, 19, 21, 22, 38, 39, 43, 295
module-name (R1404), 35, 123, 296
module-nature (R1410), 297
module-subprogram (R1408), 36, 296, 333
module-stmt (R1405), 35, 296
module-subprogram-part (R1407), 35, 83, 87, 296, 299, 580
modulo, 32, 410
MODULD= specifier, 140
MOVE_ALLOC, 143, 208, 341, 411

name, 17, 46, 50, 513
name (R603), 28, 47, 50, 51, 98, 119, 131, 204, 243, 311, 331
name association, 6, 6, 47, 517, 523
name-value subsequence, 289, 290
NAME= specifier, 97, 101, 115, 256, 257, 259, 311, 319, 330, 509
named constant, 9, 23, 44, 46, 50, 58, 62, 64, 66, 106, 107, 110, 113, 116, 118, 126, 132, 335
named-constant (R606), 51, 51, 55, 64, 91, 118, 518
named-constant-def (R852), 118, 118, 518
NAMED= specifier, 256, 259

namelist formatting, 245, 289–294

NAMER= specifier, 256, 259
numeric conversion, 171
numeric editing, 273
numeric intrinsic operation, 156
numeric sequence type, 20, 71, 126–128, 526, 529
numeric storage unit, 21, 21, 128, 443, 524, 529, 530
numeric-expr (R1027), 37, 125

NAMELIST statement, 125, 186, 290, 298
namelist-group-name, 125, 234–236, 242–244, 267, 290, 293, 298, 518, 532

O
O edit descriptor, 279
object, 10, 10, 42–45
object designator, 11, 43, 44, 112, 116, 132, 166, 289, 290

object-name (R804), 98, 98, 114, 115, 119, 120, 131, 138, 139, 518

obsolescent feature, 29, 30, 33, 542–544

octal-constant (R766), 92, 92

ONLY, 123, 123, 124, 297, 297, 298, 520, 572, 574

only (R1412), 297, 298

only-use-name (R1413), 297, 298

OPEN statement, 32, 220, 221, 225–227, 228, 228, 232, 236, 244, 245, 249, 259, 262, 279, 291, 529, 532, 537, 538, 565, 567–569

open-stmt (R1204), 37, 228, 337

OPENED= specifier, 256, 259

operator, 17

operation, 57
defined, 11, 19, 81, 153, 162, 163–166, 191, 301, 307, 313, 326, 337
elemental, 12, 155, 166, 179
intrinsic, 155–162
logical, 159
numeric, 156
relational, 160

OPERATOR, 57, 82, 162, 297, 304, 307, 575

operator, 17, 51
character, 151
defined binary, 152
defined unary, 150
elemental, 12, 155, 448
logical, 152
numeric, 150
relational, 151

operator precedence, 153

OPTIONAL attribute, 110, 110, 112, 118, 166, 185, 191, 303

optional dummy argument, 323

OPTIONAL statement, 118, 186

optional-stmt (R851), 37, 118

or-op (R1020), 51, 152, 152

or-operand (R1015), 152, 152

other-specification-stmt (R513), 36, 36

OUT, 108

OUT_OF_RANGE, 416

output statement, 233

output-item (R1217), 233, 234, 239, 239, 251, 256

OUTPUT_UNIT, 226, 227, 231, 247, 444

override, 78, 87, 97, 98, 121, 245, 273, 526

P

P edit descriptor, 284

PACK, 416

pad mode, 231

PAD= specifier, 32, 229, 231, 234, 235, 238, 249, 256, 259

padding, 343, 343, 395, 422

PARAMETER attribute, 9, 44, 91, 99, 110, 110, 118, 132

PARAMETER statement, 40, 118, 120, 300

parameter-stmt (R851), 36, 118, 518

parent component, 6, 9, 80, 84, 86, 89, 526, 554

parent data transfer statement, 238, 247, 245–249, 265, 288

parent team, 22, 46, 139, 140, 213, 214, 384, 389, 394, 415, 431, 434, 435, 444

parent type, 9, 23, 70, 74, 80, 84–87, 309, 554

parent-identifier (R1418), 299, 299

parent-string (R909), 103, 132, 132

parent-submodule-name, 299

parent-type-name, 69, 70

PARENT_TEAM, 388, 389, 444

parentheses, 164

PARITY, 417

part-name, 7, 133–135, 139

part-ref (R912), 103, 116, 126, 132, 133, 133–135, 137, 139

partially associated, 525

PASS attribute, 75, 77, 82, 313

passed-object dummy argument, 17, 77, 82, 87, 310, 315, 316, 590

PAUSE statement, 541

pending affector, 100, 237, 242, 511, 512

PENDING= specifier, 256, 257, 259

POINTER, 74, 75, 77

pointer, 6, 11, 12, 17, 19, 20, 22, 24, 46, 71, 77, 139, 144–146, 168, 302, 303, 318, 337, 432, 483, 495, 521, 586

procedure, 487

pointer assignment, 17, 105, 107, 144, 172, 174, 175, 323, 522

pointer assignment statement, 17, 22, 58, 77, 90, 165, 174, 175, 181, 352, 358

pointer association, 6, 6, 12, 22, 23, 44, 47, 85, 87, 90, 103, 108, 110, 112, 113, 133, 145, 147, 174–
pointer association context, 108, 111, 337
pointer association status, 521
POINTER statement, 119, 300
pointer-assignment-stmt (R1033), 37, 111, 174, 180, 181, 337, 533
pointer-decl (R854), 119, 119
pointer-object (R940), 144, 144, 533
pointer-stmt (R853), 37, 119, 518
polymorphic, 17, 31, 60, 61, 77, 90, 106, 107, 133, 144, 164, 169, 170, 175, 184, 185, 191, 203, 204, 239, 245, 302, 303, 313, 316–320, 336, 337, 338, 407, 411, 422, 424, 432, 521, 527
POPCNT, 417
POPPAR, 418
POS= specifier, 222–224, 234, 235, 238, 238, 256, 260, 538
position edit descriptor, 282
position-edit-desc (R1315), 269, 269
position-spec (R1227), 253, 253
POSITION= specifier, 228, 229, 231, 256, 260, 567
positional arguments, 341
potential subobject component, 9, 69, 70, 141, 337
power-op (R1007), 51, 150, 150
pre-connection, 526
precedence of operators, 153
PRECISION, 62, 418, 468
preconnected, 17, 221, 226–229, 236, 242, 441, 442, 444
preconnection, 228
prefix (R1526), 329, 329–332
prefix-spec (R1527), 329, 329, 330, 336, 338
PRESENT, 60, 75, 110, 166, 232, 342, 418, 587
present, 323
primary, 149
primary (R1001), 149, 149, 150, 335
PRINT statement, 221, 230, 233, 242, 247, 248, 252
print-stmt (R1212), 37, 234, 337
PRIVATE attribute, 71, 87, 100, 100, 113, 125, 337, 573
PRIVATE statement, 80, 81, 83, 114, 298
private-components-stmt (R745), 70, 80, 80
private-or-sequence (R729), 69, 70, 70
proc-attr-spec (R1514), 310, 311, 311
proc-component-attr-spec (R742), 75, 75, 77
proc-component-def-stmt (R741), 74, 75, 75
proc-component-ref (R1039), 175, 175, 313, 323
proc-decl (R1515), 75, 78, 310, 311, 311
proc-entity-name, 119
proc-interface (R1513), 75, 310, 310, 311
proc-language-binding-spec (R1528), 330, 330–332, 335, 493
proc-pointer-init (R1517), 311, 311
proc-pointer-name (R858), 119, 119, 144, 175
proc-pointer-object (R1038), 174, 175, 175, 181, 358, 533
proc-target (R1040), 88–90, 111, 174, 175, 175, 181, 323, 358, 524
PROCEDURE, 75, 81, 310, 333
procedure, 11, 18, 19, 47, 48, 110, 304
characteristics of, 302
dummy, 8, 12, 14, 18, 107, 120, 123, 124, 167, 175, 301, 302, 304–306, 310, 311, 314, 322, 328–330, 336, 510, 514, 519
elemental, 12, 45, 166, 175, 311, 314, 323, 327, 330, 338, 338, 341, 342
elemental, 18, 29, 38, 81, 107, 120, 123, 175, 212, 301, 302, 304–306, 310, 311, 314, 322, 328, 329, 513, 514, 518, 519, 545, 575, 580, 584
external, 14, 18, 38, 175, 301–304, 314, 322, 328, 330, 333, 358, 510, 514, 515, 519
intrinsic, 341–439
module, 18, 81, 124, 175, 301–306, 311, 314, 322, 328, 330, 333, 336, 440, 514, 515
non-Fortran, 335
type-bound, 7, 14, 17, 18, 68, 70, 77, 82, 82–87, 172, 173, 251, 297, 307, 313, 316, 317, 329, 514, 515
procedure declaration statement, 40, 107, 303, 305, 310, 335, 352, 515, 545
procedure designator, 11, 19, 45
procedure interface, 303
procedure pointer, 8, 14, 17, 17, 38, 98, 107, 108, 110, 127, 175, 176, 301, 302, 305, 306, 310, 314, 316,
SHAPE, 60, 428
shape, 20, 45, 208, 318
SHARED, 191, 194
SHIFTA, 428
SHIFTL, 428
SHIFTR, 429
SIGN, 32, 63, 429
sign (R712), 62, 62, 63, 274
sign mode, 232, 273, 284
sign-edit-desc (R1317), 269, 270
SIGN= specifier, 229, 232, 234, 235, 238, 256, 261, 284
signed-digit-string (R710), 62, 63, 273–275
signed-int-literal-constant (R707), 62, 64, 116
signed-real-literal-constant (R713), 63, 64, 116
significand (R715), 63, 63
simply contiguous, 20, 138, 139, 139, 176, 318–321, 494, 495
SIN, 429
SINH, 430
SIZE, 60, 430
size, 20, 45
size of a common block, 128
SIZE= specifier, 234, 238, 256, 261, 263, 264, 528, 531, 566
source-expr (R931), 140, 140–144, 337, 522, 523
SOURCE= specifier, 140, 142, 144, 337, 442, 529, 531, 545
SP edit descriptor, 284
SPACING, 430
special character, 49
specific interface, 15, 251, 305, 305, 306, 311, 327
specific interface block, 15, 15, 305
specific name, 20
specific-procedure (R1507), 304, 304, 306
specification, 97–129
specification expression, 20, 24, 41, 73, 75, 85, 100, 135, 166, 167, 167, 186, 334, 449, 535, 536, 543, 545
specification function, 167
specification inquiry, 166
specification-construct (R508), 36, 36, 186
specification-expr (R1028), 7, 59, 66, 99, 103, 104, 166, 166, 338
specification-part (R504), 35, 36, 36, 41, 82, 100, 113, 123, 167, 168, 295, 296, 299, 300, 304, 331–333, 336, 338
SPREAD, 431
SQRT, 31, 431, 453, 477
SS edit descriptor, 284
standard intrinsic, xviii, 16, 29, 440
standard-conforming program, 20, 29
stat-variable (R929), 41, 139, 140, 140, 142, 145–147, 209, 210, 213–215, 217, 218, 532, 536
STAT= specifier, xix, 139, 140, 143–145, 146, 189, 209, 217, 444, 445, 529, 536
STAT_FAILED_IMAGE, 41, 140, 142, 146, 147, 217, 218, 345, 384, 411, 444, 445
STAT_LOCKED, 217, 218, 444, 445
STAT_LOCKED_OTHER_IMAGE, 218, 444, 445
STAT_STOPPED_IMAGE, 142, 146, 147, 217, 345, 411, 432, 444, 445
STAT_UNLOCKED, 218, 444, 445
STAT_UNLOCKED_FAILED_IMAGE, 217, 218, 445, 445
statement, 20, 53
accessibility, 113
ALLOCATABLE, 114
ALLOCATE, 58, 59, 66, 103, 105, 140, 146, 147, 175, 208, 442, 501, 502, 521, 522, 529, 531, 536, 545
arithmetic IF, 542
ASSIGN, 541
assigned GO TO, 541
assignment, 16, 17, 44, 58, 84, 169, 181, 479, 528
ASSOCIATE, 184, 520
ASYNCHRONOUS, 114, 186, 300, 516, 519
attribute specification, 113–129
BACKSPACE, 220, 223, 249, 252, 253, 254, 566, 567
BIND, 114, 300, 509, 515
BLOCK, 46, 99, 103, 105, 186, 529
BLOCK DATA, 53, 295, 300
CALL, 22, 206, 208, 301, 313, 326, 334, 411
CASE, 198
CHANGE TEAM, 22, 40, 46, 187, 208, 217, 345, 520
CLASS DEFAULT, 203
CLASS IS, 203
CLOSE, 220, 221, 225, 227, 288, 232, 232, 249, 252, 566
COMMON, 8, 127, 127–129, 186, 298, 300, 515,
ENTRY, 12, 40, 162, 173, 296, 301, 305, 329, 331, 333, 515, 525, 526, 542, 544
ENUM, 91
ENUMERATOR, 91
EQUALIZATION, 125, 125–128, 186, 298, 300, 525, 526, 542, 544
ERROR STOP, 41, 42, 206, 536
EVENT POST, 208, 213, 214, 217, 381, 441, 442, 530, 532, 537
EVENT WAIT, 208, 213, 213, 217, 441, 442, 530, 532, 537
executable, 20, 20, 39
EXIT, 184, 194, 205
EXTERNAL, 107, 310
FAIL IMAGE, 207
file inquiry, 255
file positioning, 220, 253
FINAL, 13, 83
FLUSH, 221, 252, 255, 264
FORALL, 59, 155, 181, 516, 517, 528
FORM TEAM, 22, 40, 139, 188, 208, 214, 217, 533, 537
FORMAT, 29, 40, 52, 236, 267, 267, 296
formatted input/output, 219, 235
FUNCTION, 12, 59, 60, 121, 162, 166, 295, 329, 330, 333, 334, 515
GENERIC, 81, 82, 306, 306, 327
GO TO, 7, 53, 206, 206
IF, 155, 198
IMPLICIT, 40, 120, 125, 300
IMPLICIT NONE, 120
IMPORT, 40, 123, 519
input/output, 219–264, 528
INQUIRE, 32, 221, 222, 224, 225, 227, 228, 237, 238, 248, 249, 252, 255, 264, 265, 443, 528, 530–532, 538, 565
INTENT, 118, 186
INTERFACE, 304, 575
INTRINSIC, 300, 312
intrinsic assignment, 31, 89, 145, 147, 165, 169, 174, 175, 218, 255, 264, 291, 337, 528, 536
list-directed input/output, 236
LOCK, 208, 215, 217, 218, 443–445, 530, 532
MODULE, 295, 296
NAMELIST, 125, 186, 290, 298
namelist input/output, 236

525, 526, 542
component definition, 74
computed GO TO, 7, 206, 206, 542, 543
CONTAINS, 39, 40, 81, 335
CONTIGUOUS, 115
CONTINUE, 206, 541
CRITICAL, 163, 189, 208, 218
CYCLE, 184, 190, 193, 194, 544
DATA, 32, 40, 93, 99, 115, 128, 300, 415, 516, 519, 527, 542, 543
DEALLOCATE, 144, 146, 147, 208, 442, 502, 536
defined assignment, 31, 173, 326, 532
derived type definition, see statement, TYPE
DIMENSION, 117, 300
DO, 190, 528, 542, 544
DO CONCURRENT, 59, 180, 181, 190
DO WHILE, 190
ELSE, 197
ELSE IF, 53, 197
ELSEWHERE, 53, 178
END, 13, 40, 111, 128, 145, 146, 208, 486, 532
END ASSOCIATE, 53, 184
END BLOCK, 53, 146, 186
END BLOCK DATA, 53, 300
END CRITICAL, 53, 163, 189, 208
END DO, 53, 191
END ENUM, 53, 91
END FORALL, 53, 180
END FUNCTION, 53, 331
END IF, 53, 197, 541
END INTERFACE, 53, 304
END MODULE, 53, 296
END PROCEDURE, 53, 333
END PROGRAM, 53, 295
END SELECT, 53, 198, 203
END SUBMODULE, 53, 299
END SUBROUTINE, 53, 332
END TEAM, 40, 187, 208, 217, 345
END TYPE, 53, 70
END WHERE, 53, 178
ENDFILE, 53, 220, 221, 223, 230, 249, 252, 254, 566
nonexecutable, 20, 39
NULLIFY, 144
OPEN, 32, 220, 221, 225–227, 228, 228, 232, 236, 244, 245, 249, 259, 262, 279, 291, 529, 532, 537, 538, 565, 567–569
OPTIONAL, 118, 186
PARAMETER, 40, 118, 120, 300
PAUSE, 541
POINTER, 119, 300
pointer assignment, 17, 22, 58, 77, 90, 165, 174, 175, 181, 352, 358
PRINT, 221, 230, 233, 242, 247, 248, 252
PRIVATE, 80, 81, 83, 114, 298
PROCEDURE, 304, 306, 545
procedure declaration, 40, 107, 303, 305, 310, 335, 352, 515, 545
PROGRAM, 295
PROTECTED, 119
PUBLIC, 113, 298
RETURN, 41, 85, 111, 128, 145, 146, 189, 194, 334, 486, 532
REWIND, 220, 221, 223, 249, 252, 254, 254, 256, 566
SAVE, 119, 186, 187, 300, 515
SELECT CASE, 53, 198
SELECT RANK, 107, 201, 520
SELECT TYPE, 53, 203, 520
SELECT TYPE, 53, 203, 520
separate module subprogram, 333
SEQUENCE, 71
statement function, 12, 40, 66, 186, 296, 327, 334, 335, 516, 517, 543
STOP, 41, 42, 206, 208, 536
SUBMODULE, 295, 299
SUBROUTINE, 12, 173, 295, 329, 332, 334
SYNC ALL, 188, 208, 209, 211, 217
SYNC IMAGES, 208, 210, 217, 218
SYNC MEMORY, 208, 212, 217, 595
SYNC TEAM, 208, 213, 217
TARGET, 119, 300
TYPE, 69, 73, 100, 518
type declaration, 40, 59, 60, 78, 79, 97, 97–99, 107, 121, 125, 128, 167, 296, 300, 331, 333, 335, 336, 527
type guard, 66, 203
TYPE IS, 203
type parameter definition, 73
type-bound procedure, 81, 82
unformatted input/output, 220, 235
UNLOCK, 208, 215, 217, 218, 443, 444, 530, 532
USE, 6, 19, 40, 73, 113, 296, 300, 327, 328, 514, 518, 520, 572–574, 579
VALUE, 120, 186
VOLATILE, 120, 186, 300, 516, 519
WAIT, 227, 237, 252, 252, 569, 570
WHERE, 16, 155, 177
WRITE, 221, 226, 230, 233, 242, 247, 248, 252, 265, 528, 565, 566, 568, 569
statement entity, 20, 513, 516
statement function, 335–336, 543
statement function statement, 12, 40, 66, 186, 296, 327, 334, 335, 516, 517, 543
statement keyword, 16, 47
statement label, 7, 21, 52, 52–55, 314, 513
statement order, 39
STATUS= specifier, 228, 229, 231, 232, 233, 233, 537, 538, 568
stmt-function-stmt (R1544), 36, 296, 299, 305, 335, 518
STOP statement, 41, 42, 206, 208, 536
stop-code (R1162), 206, 206, 207
stop-stmt (R1160), 38, 85, 206
stopped image, 14, 41, 345, 411
STOPPED IMAGES, 431
storage association, 6, 6, 46, 47, 125–129, 334, 337, 432, 523–526
storage sequence, 21, 69, 71, 126–129, 300, 358, 524, 524, 525
storage unit, 21, 21, 125–129, 237, 241, 249, 252, 300, 322, 358, 524–526
character, 21, 21, 126, 128, 440, 524, 529, 530
file, 13, 21, 219, 222–225, 231, 236, 238, 244, 254, 260–262, 442, 524, 537
numeric, 21, 21, 128, 443, 524, 529, 530
unspecified, 21, 21, 524, 525, 529, 531
STORAGE_SIZE, 93, 432
stream access, 222
stream access data transfer statement, 238
stream file, 13, 21, 219, 222, 224, 262, 537
STREAM= specifier, 256, 261
stride (R922), 135, 136, 137, 138, 241
structure, 9, 21, 42, 43, 69
structure component, 21, 116, 132–134, 490, 553
ISO/IEC JTC 1/SC 22/WG5/N2118 645
structure constructor, 9, 16, 21, 43, 47, 57, 80, 88, 89, 116, 117, 164, 166, 167, 415, 443, 515, 549
structure-component (R913), 115, 116, 131, 132, 133, 139, 140, 144
structure-constructor (R756), 21, 88, 89, 116, 149, 337
subcomponent, 9, 10, 78, 88, 175, 522, 523, 527, 529, 531
submodule, 16, 18, 21, 21, 22, 38, 39, 43, 299
submodule (R1416), 35, 299
submodule identifier, 299
SUBMODULE statement, 295, 299
submodule-name, 299
submodule-stmt (R1417), 35, 299, 299
subobject, 5, 9, 10, 115, 116, 131, 133, 139, 140, 144
submodule (R1416), 35, 299
submodule-stmt (R1417), 35, 299, 299
subobject, 5, 9, 10, 21, 43–45, 108, 133, 317, 522
submodule, 16, 21, 38, 40, 41, 43, 120
subprogram, 16, 21, 38, 40, 41, 43, 120
subroutine, 22
atomic, 22, 41, 208, 209, 341, 344, 346, 360–364, 381, 440, 444, 532
collective, 22, 341, 345, 346, 369–372, 384, 432, 444
subroutine reference, 326
SUBROUTINE statement, 12, 173, 295, 329, 332, 334
subroutine-name, 305, 332, 515
subroutine-stmt (R1535), 35, 304, 305, 330, 332, 332, 515, 518
subroutine-subprogram (R1534), 21, 35, 36, 296, 332, 333
subroutines
intrinsic, 341
section, 137
subsection (R919), 116, 135, 135, 138, 241
subsection triplet, 137
subsection-triplet (R921), 135, 135–138
string, 132
substring (R908), 125, 126, 131, 132
substring ending point., 132
substring starting point, 132
substring-range (R910), 103, 132, 132, 134–136, 139, 241
suffix (R1532), 331, 331, 333
SUM, 432
SYNC ALL statement, 188, 208, 209, 211, 217
SYNC IMAGES statement, 208, 210, 217, 218
SYNC MEMORY statement, 208, 212, 217, 595
SYNC TEAM statement, 208, 213, 217
SYNC ALL statement, 188, 208, 210, 217, 218
SYNC IMAGES statement, 208, 210
SYNC MEMORY statement, 208, 212
SYNC TEAM statement, 208, 213, 217
synchronized execution, 188
synchronous input/output, 230, 236, 238, 241
SYSTEM_CLOCK, 433
T
T edit descriptor, 283
TAN, 434
TANH, 434
TARGET statement, 119, 300
target-decl (R860), 119, 119
target-stmt (R859), 37, 119, 518
team, 14, 22, 22, 40, 45, 46, 139, 140, 146, 188, 210, 213, 214, 217, 341, 345, 394, 415, 432, 435, 537
team number, 22, 139, 214
team variable, 25, 188, 445
team-construct-name, 187
team-number (R1176), 214, 214
team-variable (R1115), 139, 187, 187, 188, 213, 214, 217, 533
TEAM= specifier, 139, 139
TEAM_NUMBER, 434
TEAM_NUMBER= specifier, 139, 139
TEAM_TYPE, 25, 187, 384, 389, 394, 415, 431, 434, 435, 445
THEN, 197
THIS_IMAGE, 168, 435
TINY, 430, 431, 435
TKR compatible, 309
TL edit descriptor, 283
totally associated, 525
TR edit descriptor, 283
TRAILZ, 436
TRANSFER, 168, 436
transfer of control, 183, 206, 263, 264
transformational function, 22, 168, 335, 341, 341, 342, 346, 365, 366, 440, 441, 453, 454
TRANSPOSE, 437
TRIM, 437
truncation, 343, 395, 422
TYPE, 59
type, 22, 42, 57–94
abstract, 23, 59, 82, 85, 85, 88, 133, 141
complex, 64–65
declared, 23, 60, 61, 77, 89, 90, 93, 94, 97, 133, 134, 141, 142, 144, 162, 164, 169, 172, 173, 175, 176, 184, 203, 204, 251, 308, 313, 316, 319, 329, 336, 383, 407, 411, 424, 441, 443, 451
derived, 11, 21, 23, 42, 43, 46, 57, 68–90, 93, 490, 491
dynamic, 17, 23, 24, 60, 61, 84, 85, 87, 90, 94, 113, 142, 144, 146, 162, 164, 170, 172, 173, 175, 176, 185, 203, 204, 208, 212, 251, 313, 319, 320, 329, 351, 383, 407, 411, 424, 432, 517, 521, 527, 554, 603
expression, 164
extended, 6, 9, 14, 23, 23, 74, 80, 84–86, 526, 546, 551
extensible, 23, 59, 70, 77, 85, 245, 383, 424, 554, 592
extension, 23, 60, 85, 87, 203, 204, 319, 383, 592
integer, 61–62
intrinsic, 9, 23, 42, 43, 57, 61–68
logical, 68
operation, 165
parent, 9, 23, 70, 74, 80, 84–87, 309, 554
primary, 164
real, 62–64, 64
type compatible, 23, 60, 78, 141, 169, 174, 309, 317, 411
type conformance, 169
type declaration statement, 40, 59, 60, 78, 79, 97, 97–99, 107, 121, 125, 128, 167, 296, 300, 331, 333, 335, 336, 527
type equality, 71
type guard statement, 66, 203
TYPE IS statement, 203
type parameter definition statement, 73
type parameter inquiry, 24, 134, 164, 167
type parameter keyword, 16, 47, 88
type parameter order, 24, 73, 74
type specifier, 59
character, 65
CLASS, 60
COMPLEX, 64
derived type, 59
DOUBLE PRECISION, 63
INTEGER, 61
LOGICAL, 68
REAL, 63
TYPE, 59
TYPE statement, 69, 73, 100, 518
type-attr-spec (R728), 69, 69, 85
type-bound procedure, 7, 14, 17, 18, 68, 70, 77, 82, 82–87, 172, 173, 251, 297, 307, 313, 316, 317, 329, 514, 515
type-bound procedure statement, 81, 82
type-bound-generic-stmt (R751), 81, 82, 82, 307
type-bound-proc-binding (R748), 81, 81
type-bound-proc-decl (R750), 81, 81
type-bound-procedure-part (R746), 69, 71, 81, 83, 491
type-bound-procedure-stmt (R749), 81, 81
type-declaration-stmt (R801), 36, 66, 97, 97, 336, 518
type-guard-stmt (R1154), 203, 203
type-name, 69, 70, 72, 82, 88
type-param-attr-spec (R734), 73, 73
type-param-decl (R733), 73, 73
type-param-def-stmt (R732), 69, 73, 73
type-param-inquiry (R916), 24, 134, 134, 149, 515
type-param-name, 69, 73, 75, 134, 149, 515, 518
type-param-spec (R755), 88, 88
type-param-value (R701), 24, 58, 58, 59, 65, 66, 75, 88, 141, 142, 544
type-spec (R702), 59, 59, 66, 93, 94, 140–143, 203

U
UBOUND, 60, 202, 318, 437
UCOBOUND, 438
ultimate argument, 24, 142, 146, 316, 317, 321, 324, 345
ultimate entity, 298
undefined, 12, 24, 44, 145, 522, 527, 528
undefined of variables, 527
underflow mode, 451, 452, 454, 459, 471, 478, 479, 540
underscore (R602), 49, 49
UNFORMATTED, 245, 246, 304
unformatted data transfer, 244
unformatted input/output statement, 220, 235
unformatted record, 219
UNFORMATTED= specifier, 256, 262
Unicode file, 230
UNIT= specifier, 228, 233, 234, 252, 253, 255, 256
unlimited polymorphic, 24, 60, 60, 93, 127, 141, 174, 203, 319, 383, 424, 432, 603
unlimited-format-item (R1305), 267, 268, 268, 271
UNLOCK statement, 208, 215, 217, 218, 443, 444, 530, 532
unlock-stmt (R1180), 38, 215
unordered segments, 208, 209, 344, 352, 381, 420
UNPACK, 438
unsaved, 24, 144, 145, 332, 522, 523, 529, 530, 532
unspecified storage unit, 21, 21, 524, 525, 529, 531
until-spec (R1174), 213, 213
UNTIL_COUNT= specifier, 213, 381
upper-bound (R818), 104, 104, 105
upper-bound-expr (R936), 140, 140, 141, 174
upper-cobound (R813), 102, 103, 103
use association, 6, 6, 17, 39, 47, 60, 66, 86, 100, 111, 113, 114, 121, 125–127, 166, 167, 175, 296, 295–299, 305, 333, 337, 516–518, 521
use path, 298
USE statement, 6, 19, 40, 73, 113, 296, 300, 327, 328, 514, 518, 520, 572–574, 579
use-defined-operator (R1415), 297, 297, 298

V
v (R1312), 248, 269, 269, 282
value separator, 285
VALUE statement, 120, 186
value-stmt (R861), 37, 120
variable, 10, 21, 24, 43, 44, 46, 50, 110
definition & undefinition, 527
variable (R902), 89, 115, 116, 131, 131, 139, 169–172, 174, 175, 179, 181, 184, 187, 203, 213, 215, 239, 313, 323, 479, 532
variable-name (R903), 125, 127, 131, 131, 132, 140, 144, 174, 191, 518, 532
vector subscript, 25, 45, 78, 103, 133, 137, 138, 184, 193, 225, 289, 318, 319, 324, 521, 523, 532, 587
vector-subscript (R923), 135, 136, 136, 137
VERIFY, 439
VOLATILE attribute, 30, 31, 112, 112, 113, 120, 174, 176, 185, 194, 296, 298, 302, 303, 319, 321, 336, 518, 519, 524, 529, 532, 552
VOLATILE statement, 120, 186, 300, 516, 519
volatile-stmt (R862), 37, 120

W
w (R1308), 269, 269, 273–282, 286, 288, 291
wait operation, 228, 232, 238, 241, 242, 252, 252–253, 255, 260, 263
WAIT statement, 227, 237, 252, 252, 569, 570
wait-spec (R1223), 252, 252
wait-stmt (R1222), 38, 252, 337
WHERE construct, 16, 177
WHERE statement, 16, 155, 177
where-assignment-stmt (R1045), 155, 177, 178, 178, 179, 181
where-body-construct (R1044), 177, 178, 178, 179
where-construct (R1042), 37, 177, 178, 180, 181
where-construct-name, 178
where-construct-stmt (R1043), 7, 177, 178, 178, 181, 206
where-stmt (R1041), 38, 177, 178, 180, 181
WHILE, 190, 192, 193
whole array, 25, 135, 135, 136, 398, 437
WRITE (FORMATTED), 245, 246, 304
WRITE (UNFORMATTED), 245, 246, 304
WRITE statement, 221, 226, 230, 233, 242, 247, 248, 252, 265, 528, 565, 566, 568, 569
write-stmt (R1211), 38, 233, 234, 337, 532
WRITE= specifier, 257, 262

X
X edit descriptor, 283
xyz-list (R401), 28
xyz-name (R402), 28

Z
Z edit descriptor, 279
zero-size array, 45, 105, 117