WD 1539-1

J3/15-007
(F2015 Working Document)

24th December 2014 11:24

This is an internal working document of J3 and WG5.

NOTE: This working document is only available as a PDF file.

This page intentionally left nonblank.

Contents

(Blank page)

List of Tables

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.*

6 This fourth edition cancels and replaces the third edition (ISO/IEC 1539-1:2010), which has been technically revised. It also incorporates the Technical Corrigenda ISO/IEC 1539-1:2010/Cor. 1:2012 and ISO/IEC 1539-1:2010/Cor. 2:2013, and the Technical Specification ISO/IEC TS 29113:2012.

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*

10 — *Part 3: Conditional compilation*

Introduction

1 This part of ISO/IEC 1539 comprises the specification of the base Fortran language, informally known as Fortran 2015. With the limitations noted in 1.6.4, 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 part of ISO/IEC 1539.

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

- Data usage and computation:
  Labeled DO loops have been redundant since Fortran 90 and are now specified to be obsolescent. The arithmetic IF statement has been deleted. The EQUIVALENCE and COMMON statements and the block data program unit have been redundant since Fortran 90 and are now specified to be obsolescent. The nonblock DO construct has been deleted. The FORALL is now specified to be obsolescent. The type and kind of an implied DO variable in an array constructor or DATA statement can be specified within the constructor or statement.
- 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 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.
- 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 of 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 obsolescent.
- 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 ERROR STOP statement can appear in a pure subprogram. 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 that 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 RECURSIVE keyword is advisory only. The NON_RECURSIVE keyword specifies that a procedure is not recursive.
- Features previously described by ISO/IEC TS 29113:2012:
  A dummy data object can assume its rank from its effective argument. A dummy data object can assume

the type from its effective argument, without having the ability to perform type selection. An interoperable procedure can have dummy arguments that are assumed-type and/or assumed-rank. An interoperable procedure can have dummy data objects that are allocatable, assumed-shape, optional, or pointers. The character length of a dummy data object of an interoperable procedure can be assumed.

- 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 standard intrinsic relational operations on IEEE numbers provide the compareSignalingrelation operations. 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_QUIET_EQ, IEEE_QUIET_GE, IEEE_QUIET_GT, IEEE_QUIET_LE, IEEE_QUIET_LT, and IEEE_QUIET_NE perform quiet 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_REAL performs rounded conversions to real type. The function IEEE_RINT now has a ROUND argument to perform specific rounding. The new function IEEE_SIGNBIT tests the sign bit of an IEEE number.

3 This part of ISO/IEC 1539 is organized in 16 clauses, dealing with 8 conceptual areas. These 8 areas, and the clauses in which they are treated, are:

High/low level concepts	Clauses 1, 2, 3
Data concepts	Clauses 4, 5, 6
Computations	Clauses 7, 13, 14
Execution control	Clause 8
Input/output	Clauses 9, 10
Program units	Clauses 11, 12
Interoperability with C	Clause 15
Scoping and association rules	Clause 16

4 It also contains the following nonnormative material:

Processor dependencies	Annex A
Deleted and obsolescent features	Annex B
Extended notes	Annex C
Index	Index

Information technology — Programming languages — Fortran —

Part 1:
Base language

1 Overview

1.1 Scope

1 This part of ISO/IEC 1539 specifies the form and establishes the interpretation of programs expressed in the base Fortran language. The purpose of this part of ISO/IEC 1539 is to promote portability, reliability, maintainability, and efficient execution of Fortran programs for use on a variety of computing systems.

2 This part of ISO/IEC 1539 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 part of ISO/IEC 1539 apply to programs rather than processors.

4 This part of ISO/IEC 1539 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 part of ISO/IEC 1539 fails to establish an interpretation except for the processor detection and reporting requirements in items (2) to (10) of 1.5,
- 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 14,
- the physical properties of input/output records, files, and units, or
- the physical properties and implementation of storage.

1.2 Normative references

1 The following referenced standards are indispensable for the application 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.

ISO/IEC 646:1991 (International Reference Version), *Information technology—ISO 7-bit coded character set for information interchange*

ISO/IEC 9899:2011, *Programming languages—C*

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

ISO/IEC/IEEE 60559:2011, *Binary floating-point arithmetic for microprocessor systems*

1.3 Terms and definitions

1 For the purposes of this document, the following terms and definitions apply.

1.3.1
abstract interface
set of procedure characteristics with dummy argument names (12.4.1)

1.3.2
actual argument
entity (R1225) that appears in a procedure reference

1.3.3
allocatable
having the ALLOCATABLE attribute (5.5.3)

1.3.4
array
set of scalar data, all of the same type and type parameters, whose individual elements are arranged in a rectangular pattern

1.3.4.1
array element
scalar individual element of an array

1.3.4.2
array pointer
array with the POINTER attribute (5.5.14)

1.3.4.3
array section
array subobject designated by *array-section*, and which is itself an array (6.5.3.3)

1.3.4.4
assumed-shape array
nonallocatable nonpointer dummy argument array that takes its shape from its effective argument (5.5.8.3)

1.3.4.5
assumed-size array
dummy argument array whose size is assumed from that of its effective argument (5.5.8.5)

1.3.4.6
deferred-shape array
allocatable array or array pointer, declared with a *deferred-shape-spec-list* (5.5.8.4)

1.3.4.7
explicit-shape array
array declared with an *explicit-shape-spec-list*, which specifies explicit values for the bounds in each dimension of the array (5.5.8.2)

1.3.5
ASCII character

character whose representation method corresponds to ISO/IEC 646:1991 (International Reference Version)

1.3.6
associate name

name of construct entity associated with a selector of an ASSOCIATE or SELECT TYPE construct (8.1.3)

1.3.7
associating entity

⟨in a dynamically-established association⟩ the entity that did not exist prior to the establishment of the association (16.5.5)

1.3.8
association

inheritance association (16.5.4), name association (16.5.1), pointer association (16.5.2), or storage association (16.5.3).

1.3.8.1
argument association

association between an effective argument and a dummy argument (12.5.2)

1.3.8.2
construct association

association between a selector and an associate name in an ASSOCIATE or SELECT TYPE construct (8.1.3, 8.1.9, 16.5.1.6)

1.3.8.3
host association

name association, other than argument association, between entities in a submodule or contained scoping unit and entities in its host (16.5.1.4)

1.3.8.4
inheritance association

association between the inherited components of an extended type and the components of its parent component

1.3.8.5
linkage association

association between a variable or common block with the BIND attribute and a C global variable (15.9, 16.5.1.5)

1.3.8.6
name association

argument association, construct association, host association, linkage association, or use association (16.5.1)

1.3.8.7
pointer association

association between a pointer and an entity with the TARGET attribute (16.5.2)

1.3.8.8
storage association

association between storage sequences (16.5.3)

1.3.8.9
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 (11.2.2)

1.3.9
assumed-rank dummy data object
dummy data object that assumes the rank, shape, and size of its effective argument

1.3.10
assumed-type
declared with a TYPE(*) type specifier (4.3.2)

1.3.11
attribute
property of an entity that determines its uses (5.1)

1.3.12
automatic data object
automatic object
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

1.3.13
base object
⟨*data-ref*⟩ object designated by the leftmost *part-name* (6.4.2)

1.3.14
binding
type-bound procedure or final subroutine (4.5.5)

1.3.15
binding name
name given to a specific or generic type-bound procedure in the type definition (4.5.5)

1.3.16
binding label
default character value specifying the name by which a global entity with the BIND attribute is known to the companion processor (15.10.2, 15.9.2)

1.3.17
block
sequence of executable constructs formed by the syntactic class *block* and which is treated as a unit by the executable constructs described in 8.1

1.3.18
bound
array bound
limit of a dimension of an array

1.3.19
branch target statement
action-stmt, *associate-stmt*, *end-associate-stmt*, *if-then-stmt*, *end-if-stmt*, *select-case-stmt*, *end-select-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*, or *where-construct-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 (8.2.1)

1.3.20
C address
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 1.1

This is the concept that ISO/IEC 9899:2011 calls the address.

1.3.21
C descriptor
C structure of type CFI_cdesc_t defined in the source file ISO_Fortran_binding.h (15.4, 15.5)

1.3.22
character context
within a character literal constant (4.4.4) or within a character string edit descriptor (10.3.2)

1.3.23
characteristics
⟨dummy argument⟩ being a dummy data object, dummy procedure, or an asterisk (alternate return indicator)

1.3.24
characteristics
⟨dummy data object⟩ properties listed in 12.3.2.2

1.3.25
characteristics
⟨dummy procedure or dummy procedure pointer⟩ properties listed in 12.3.2.3

1.3.26
characteristics
⟨function result⟩ properties listed in 12.3.3

1.3.27
characteristics
⟨procedure⟩ properties listed in 12.3.1

1.3.28
coarray
data entity that has nonzero corank (2.4.7)

1.3.29
cobound
bound (limit) of a codimension

1.3.30
codimension
dimension of the pattern formed by a set of corresponding coarrays

1.3.31
coindexed object
data object whose designator includes an *image-selector* (R624, 6.6)

1.3.32
collating sequence
one-to-one mapping from a character set into the nonnegative integers (4.4.4.4)

1.3.33
common block
block of physical storage specified by a COMMON statement (5.9.2)

1.3.33.1
blank common
unnamed common block

1.3.34
companion processor
processor-dependent mechanism by which global data and procedures may be referenced or defined (2.5.7)

1.3.35
component
part of a derived type, or of an object of derived type, defined by a *component-def-stmt* (4.5.4)

1.3.35.1
direct component
one of the components, or one of the direct components of a nonpointer nonallocatable component (4.5.1)

1.3.35.2
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 (4.5.7.2)

1.3.35.3
potential subobject component
nonpointer component, or potential subobject component of a nonpointer component (4.5.1)

1.3.35.4
subcomponent
⟨structure⟩ direct component that is a subobject of the structure (6.4.2)

1.3.35.5
ultimate component
component that is of intrinsic type, a pointer, or allocatable; or an ultimate component of a nonpointer nonallocatable component of derived type

1.3.36
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) (4.5.4.7)

1.3.37
conformable
⟨of two data entities⟩ having the same shape, or one being an array and the other being scalar

1.3.38
connected
relationship between a unit and a file: each is connected if and only if the unit refers to the file (9.5.4)

1.3.39
constant
data object that has a value and which cannot be defined, redefined, or become undefined during execution of a program (3.2.3, 6.3)

1.3.39.1
literal constant
constant that does not have a name (R305, 4.4)

1.3.39.2
named constant
named data object with the PARAMETER attribute (5.5.13)

1.3.40
construct entity
entity whose identifier has the scope of a construct (16.1, 16.4)

1.3.41
constant expression
expression satisfying the requirements specified in 7.1.12, thus ensuring that its value is constant

1.3.42
contiguous
⟨array⟩ having array elements in order that are not separated by other data objects, as specified in 5.5.7

1.3.43
contiguous
⟨multi-part data object⟩ that the parts in order are not separated by other data objects

1.3.44
corank
number of codimensions of a coarray (zero for objects that are not coarrays)

1.3.45
cosubscript
(R625) scalar integer expression in an *image-selector* (R624)

1.3.46
data entity
data object, result of the evaluation of an expression, or the result of the execution of a function reference

1.3.47
data object
object
constant (4.1.4), variable (6), or subobject of a constant (2.4.3.2.3)

1.3.48
decimal symbol
character that separates the whole and fractional parts in the decimal representation of a real number in a file (10.6)

1.3.49
declaration
specification of attributes for various program entities

NOTE 1.2
Often this involves specifying the type of a named data object or specifying the shape of a named array object.

1.3.50
default initialization
mechanism for automatically initializing pointer components to have a defined pointer association status, and nonpointer components to have a particular value (4.5.4.6)

1.3.51
default-initialized
⟨subcomponent⟩ subject to a default initialization specified in the type definition for that component (4.5.4.6)

1.3.52
definable
capable of definition and permitted to become defined

1.3.53
defined
⟨data object⟩ has a valid value

1.3.54
defined
⟨pointer⟩ has a pointer association status of associated or disassociated (16.5.2.2)

1.3.55
defined assignment
assignment defined by a procedure (7.2.1.4, 12.4.3.5.3)

1.3.56
defined input/output
input/output defined by a procedure and accessed via a *defined-io-generic-spec* (R1209, 9.6.4.8)

1.3.57
defined operation
operation defined by a procedure (7.1.6.1, 12.4.3.5.2)

1.3.58
definition
⟨data object⟩ process by which the data object becomes defined (16.6.5)

1.3.59
definition
⟨derived type (4.5.2), enumeration (4.6), or procedure (12.6)⟩ specification of the type, enumeration, or procedure

1.3.60
descendant
⟨module or submodule⟩ submodule that extends that module or submodule or that extends another descendant thereof

1.3.61
designator
name followed by zero or more component selectors, complex part selectors, array section selectors, array element selectors, image selectors, and substring selectors (6.1)

1.3.61.1
complex part designator
designator that designates the real or imaginary part of a complex data object, independently of the other part (6.4.4)

1.3.61.2
object designator
data object designator
designator for a data object

NOTE 1.3

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

1.3.61.3
procedure designator
designator for a procedure

1.3.62
disassociated
⟨pointer association⟩ pointer association status of not being associated with any target and not being undefined (16.5.2.2)

1.3.63
disassociated
⟨pointer⟩ has a pointer association status of disassociated

1.3.64
dummy argument
entity whose identifier appears in a dummy argument list (R1237) 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

1.3.64.1
dummy data object
dummy argument that is a data object

1.3.64.2
dummy function
dummy procedure that is a function

1.3.65
effective argument
entity that is argument-associated with a dummy argument (12.5.2.3)

1.3.66
effective item
scalar object resulting from the application of the rules in 9.6.3 to an input/output list

1.3.67
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 1.4

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

1.3.67.1
elemental assignment
assignment that operates elementally

1.3.67.2
elemental operation
operation that operates elementally

1.3.67.3
elemental operator
operator in an elemental operation

1.3.67.4
elemental procedure
elemental intrinsic procedure or procedure defined by an elemental subprogram

1.3.67.5
elemental reference
reference to an elemental procedure with at least one array actual argument

1.3.67.6
elemental subprogram
subprogram with the ELEMENTAL prefix

1.3.68
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*

1.3.69
explicit initialization
initialization of a data object by a specification statement (5.4, 5.6.7)

1.3.70
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 (12.4.2)

1.3.71
extent
number of elements in a single dimension of an array

1.3.72
external file
file that exists in a medium external to the program (9.3)

1.3.73
external unit
external input/output unit
entity that can be connected to an external file

1.3.74
file storage unit
unit of storage in a stream file or an unformatted record file (9.3.5)

1.3.75
final subroutine
subroutine whose name appears in a FINAL statement (4.5.6) in a type definition, and which can be automatically invoked by the processor when an object of that type is finalized (4.5.6.2)

1.3.76
finalizable
⟨type⟩ has a final subroutine or a nonpointer nonallocatable component of finalizable type

1.3.77
finalizable
⟨nonpointer data entity⟩ of finalizable type

1.3.78
finalization
process of calling final subroutines when one of the events listed in 4.5.6.3 occurs

1.3.79
function
procedure that is invoked by an expression

1.3.80
function result
entity that returns the value of a function

1.3.81
generic identifier
lexical token that identifies a generic set of procedures, intrinsic operations, and/or intrinsic assignments

1.3.82
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 (12.6.2.4)

1.3.83
host scoping unit
host
scoping unit immediately surrounding another scoping unit, or the scoping unit extended by a submodule

1.3.84
IEEE infinity
ISO/IEC/IEEE 60559:2011 conformant infinite floating-point value

1.3.85
IEEE NaN
ISO/IEC/IEEE 60559:2011 conformant floating-point datum that does not represent a number

1.3.86
image
instance of a Fortran program (2.3.4)

1.3.87
image index
integer value identifying an image

1.3.88
image control statement
statement that affects the execution ordering between images (8.5)

1.3.89
implicit interface
interface of a procedure that includes only the type and type parameters of a function result (12.4.2, 12.4.3.9)

1.3.90
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 1.5
That is, inclusive scope is the scope as if BLOCK constructs were not scoping units.

1.3.91
inherit
⟨extended type⟩ acquire entities (components, type-bound procedures, and type parameters) through type extension from the parent type

1.3.92
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

1.3.93
interface
⟨procedure⟩ name, procedure characteristics, dummy argument names, binding label, and generic identifiers (12.4.1)

1.3.93.1
generic interface
set of procedure interfaces identified by a generic identifier

1.3.93.2
specific interface
interface identified by a nongeneric name

1.3.94
interface block
abstract interface block, generic interface block, or specific interface block (12.4.3.2)

1.3.94.1
abstract interface block
interface block with the ABSTRACT keyword; collection of interface bodies that specify named abstract interfaces

1.3.94.2
generic interface block
interface block with a *generic-spec*; collection of interface bodies and procedure statements that are to be given that generic identifier

1.3.94.3
specific interface block
interface block with no *generic-spec* or ABSTRACT keyword; collection of interface bodies that specify the interfaces of procedures

1.3.95
interoperable
⟨Fortran entity⟩ equivalent to an entity defined by or definable by the companion processor (15.3)

1.3.96
intrinsic
type, procedure, module, assignment, operator, or input/output operation defined in this part of ISO/IEC 1539 and accessible without further definition or specification, or a procedure or module provided by a processor but not defined in this part of ISO/IEC 1539

1.3.96.1
standard intrinsic
⟨procedure or module⟩ defined in this part of ISO/IEC 1539 (13)

1.3.96.2
nonstandard intrinsic
⟨procedure or module⟩ provided by a processor but not defined in this part of ISO/IEC 1539

1.3.97
internal file
character variable that is connected to an internal unit (9.4)

1.3.98
internal unit
input/output unit that is connected to an internal file (9.5.4)

1.3.99
ISO 10646 character
character whose representation method corresponds to UCS-4 in ISO/IEC 10646

1.3.100
keyword
statement keyword, argument keyword, type parameter keyword, or component keyword

1.3.100.1
argument keyword
word that identifies the corresponding dummy argument in an actual argument list

1.3.100.2
component keyword
word that identifies a component in a structure constructor

1.3.100.3
statement keyword
word that is part of the syntax of a statement (2.5.2)

1.3.100.4
type parameter keyword
word that identifies a type parameter in a type parameter list

1.3.101
lexical token
keyword, name, literal constant other than a complex literal constant, operator, label, delimiter, comma, =, =>, :, ::, ;, or % (3.2)

1.3.102
line
sequence of zero or more characters

1.3.103
main program
program unit that is not a subprogram, module, submodule, or block data program unit (11.1)

1.3.104
masked array assignment
assignment statement in a WHERE statement or WHERE construct (7.2.3)

1.3.105
module
program unit containing (or accessing from other modules) definitions that are to be made accessible to other program units (11.2)

1.3.106
name
identifier of a program consituent, formed according to the rules given in 3.2.2

1.3.107
NaN
Not a Number, a symbolic floating-point datum (ISO/IEC/IEEE 60559:2011)

1.3.108
operand
data value that is the subject of an operator

1.3.109
operator
intrinsic-operator, *defined-unary-op*, or *defined-binary-op* (R308, R703, R723)

1.3.110
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 (4.5.4.5)

1.3.111
pointer
data pointer (1.3) or procedure pointer (1.3)

1.3.111.1
data pointer
data entity with the POINTER attribute (5.5.14)

1.3.111.2
procedure pointer
procedure with the EXTERNAL and POINTER attributes (5.5.9, 5.5.14)

1.3.112
pointer assignment
association of a pointer with a target, by execution of a pointer assignment statement (7.2.2) or an intrinsic assignment statement (7.2.1.2) for a derived-type object that has the pointer as a subobject

1.3.113
polymorphic
⟨data entity⟩ able to be of differing dynamic types during program execution (4.3.2.3)

1.3.114
preconnected
⟨file or unit⟩ connected at the beginning of execution of the program (9.5.5)

1.3.115
procedure
entity encapsulating an arbitrary sequence of actions that can be invoked directly during program execution

1.3.115.1
dummy procedure
procedure that is a dummy argument (12.2.2.3)

1.3.115.2
external procedure
procedure defined by an external subprogram (R203) or by means other than Fortran (12.6.3)

1.3.115.3
internal procedure
procedure defined by an internal subprogram (R211)

1.3.115.4
module procedure
procedure that is defined by a module subprogram (R1108)

1.3.115.5
pure procedure
procedure declared or defined to be pure according to the rules in 12.7

1.3.115.6
type-bound procedure
procedure that is bound to a derived type and referenced via an object of that type (4.5.5)

1.3.116
processor
combination of a computing system and mechanism by which programs are transformed for use on that computing system

1.3.117
processor dependent
not completely specified in this part of ISO/IEC 1539, having methods and semantics determined by the processor

1.3.118
program
set of Fortran program units and entities defined by means other than Fortran that includes exactly one main program

1.3.119
program unit
main program, external subprogram, module, submodule, or block data program unit (2.2.1)

1.3.120
rank
number of array dimensions of a data entity (zero for a scalar entity)

1.3.121
record
sequence of values or characters in a file (9.2)

1.3.122
record file
file composed of a sequence of records (9.1)

1.3.123
reference
data object reference, procedure reference, or module reference

1.3.123.1
data object reference
appearance of a data object designator (6.1) in a context requiring its value at that point during execution

1.3.123.2
function reference
appearance of the procedure designator for a function, or operator symbol in a context requiring execution of the function during expression evaluation (12.5.3)

1.3.123.3
module reference
appearance of a module name in a USE statement (11.2.2)

1.3.123.4
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 (10.7.6) or derived-type finalization (4.5.6.2)

1.3.124
saved
having the SAVE attribute (5.5.16)

1.3.125
scalar
data entity that can be represented by a single value of the type and that is not an array (6.5)

1.3.126
scoping unit
BLOCK construct, derived-type definition, interface body, program unit, or subprogram, excluding all nested scoping units in it

1.3.126.1
block scoping unit
scoping unit of a BLOCK construct

1.3.127
sequence
set of elements ordered by a one-to-one correspondence with the numbers 1, 2, to n

1.3.128
sequence structure
scalar data object of a sequence type (4.5.2.3)

1.3.129
sequence type
derived type with the SEQUENCE attribute (4.5.2.3)

1.3.129.1
character sequence type
sequence type with no type parameters, no allocatable or pointer components, and whose components are all default character or of another character sequence type

1.3.129.2
numeric sequence type
sequence type with no type parameters, 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

1.3.130
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 1.6**
> Thus the shape of a scalar data entity is an array with rank one and size zero.

1.3.131
simply contiguous
⟨array designator or variable⟩ satisfying the conditions specified in 6.5.4

> **NOTE 1.7**
> These conditions are simple ones which make it clear that the designator or variable designates a contiguous array.

1.3.132
size
⟨array⟩ total number of elements in the array

1.3.133
specification expression
expression satisfying the requirements specified in 7.1.11, thus being suitable for use in specifications

1.3.134
specific name
name that is not a generic name

1.3.135
standard-conforming program
program that uses only those forms and relationships described in, and has an interpretation according to, this part of ISO/IEC 1539

1.3.136
statement
sequence of one or more complete or partial lines satisfying a syntax rule that ends in *-stmt* (3.3)

1.3.136.1
executable statement
statement that is a member of the syntactic class *executable-construct*, excluding those in the *specification-part* of a BLOCK construct

1.3.136.2
nonexecutable statement
statement that is not an executable statement

1.3.137
statement entity
entity whose identifier has the scope of a statement or part of a statement (16.1, 16.4)

1.3.138
statement label
label
unsigned positive number of up to five digits that refers to an individual statement (3.2.5)

1.3.139
storage sequence
contiguous sequence of storage units (16.5.3.2)

1.3.140
storage unit
character storage unit, numeric storage unit, file storage unit, or unspecified storage unit (16.5.3.2)

1.3.140.1
character storage unit
unit of storage that holds a default character value (16.5.3.2)

1.3.140.2
numeric storage unit
unit of storage that holds a default real, default integer, or default logical value (16.5.3.2)

1.3.140.3
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 (16.5.3.2)

1.3.141
stream file
file composed of a sequence of file storage units (9.1)

1.3.142
structure
scalar data object of derived type (4.5)

1.3.142.1
structure component
component of a structure

1.3.142.2
structure constructor
syntax (*structure-constructor*, 4.5.10) that specifies a structure value or creates such a value

1.3.143
submodule
program unit that extends a module or another submodule (11.2.3)

1.3.144
subobject
portion of data object that can be referenced, and if it is a variable defined, independently of any other portion

1.3.145
subprogram
function-subprogram (R1229) or *subroutine-subprogram* (R1235)

1.3.145.1
external subprogram
subprogram that is not contained in a main program, module, submodule, or another subprogram

1.3.145.2
internal subprogram
subprogram that is contained in a main program or another subprogram

1.3.145.3
module subprogram
subprogram that is contained in a module or submodule but is not an internal subprogram

1.3.146
subroutine
procedure invoked by a CALL statement, by defined assignment, or by some operations on derived-type entities

1.3.146.1
atomic subroutine
intrinsic subroutine that performs an action on its ATOM argument atomically

1.3.147
target
entity that is pointer associated with a pointer (16.5.2.2), entity on the right-hand-side of a pointer assignment statement (R733), or entity with the TARGET attribute (5.5.17)

1.3.148
transformational function
intrinsic function, or function in an intrinsic module, that is neither elemental nor an inquiry function

1.3.149
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 (4.1)

1.3.149.1
abstract type
type with the ABSTRACT attribute (4.5.7.1)

1.3.149.2
declared type
type that a data entity is declared to have, either explicitly or implicitly (4.3.2, 7.1.9)

1.3.149.3
derived type
type defined by a type definition (4.5) or by an intrinsic module

1.3.149.4
dynamic type
type of a data entity at a particular point during execution of a program (4.3.2.3, 7.1.9)

1.3.149.5
extended type
type with the EXTENDS attribute (4.5.7.1)

1.3.149.6
extensible type
type that may be extended using the EXTENDS clause (4.5.7.1)

1.3.149.7
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

1.3.149.8
intrinsic type
type defined by this part of ISO/IEC 1539 that is always accessible (4.4)

1.3.149.9
numeric type
one of the types integer, real, and complex

1.3.149.10
parent type
⟨extended type⟩ type named in the EXTENDS clause

1.3.149.11
type compatible
compatibility of the type of one entity with respect to another for purposes such as argument association, pointer association, and allocation (4.3.2)

1.3.149.12
type parameter
value used to parameterize a type (4.2)

1.3.149.12.1
assumed type parameter
length type parameter that assumes the type parameter value from another entity

NOTE 1.8

The other entity is

- the selector for an associate name,
- the *constant-expr* for a named constant of type character, or
- the effective argument for a dummy argument.

1.3.149.12.2
deferred type parameter
length type parameter whose value can change during execution of a program and whose *type-param-value* is a colon

1.3.149.12.3
kind type parameter
type parameter whose value is required to be defaulted or given by a constant expression

1.3.149.12.4
length type parameter
type parameter whose value is permitted to be assumed, deferred, or given by a specification expression

1.3.149.12.5
type parameter inquiry
syntax (*type-param-inquiry*) that is used to inquire the value of a type parameter of a data object (6.4.5)

1.3.149.12.6
type parameter order
ordering of the type parameters of a type (4.5.3.2) used for derived-type specifiers (*derived-type-spec*, 4.5.9)

1.3.150
ultimate argument
nondummy entity with which a dummy argument is associated via a chain of argument associations (12.5.2.3)

1.3.151
undefined
⟨data object⟩ does not have a valid value

1.3.152
undefined
⟨pointer⟩ does not have a pointer association status of associated or disassociated (16.5.2.2)

1.3.153
unit
input/output unit
means, specified by an *io-unit*, for referring to a file (9.5.1)

1.3.154
unlimited polymorphic
able to have any dynamic type during program execution (4.3.2.3)

1.3.155
unsaved
not having the SAVE attribute (5.5.16)

1.3.156
variable
data entity that can be defined and redefined during execution of a program

1.3.156.1
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 accessible outside that scoping unit

1.3.156.2
lock variable
scalar variable of type LOCK_TYPE (13.8.2.16) from the intrinsic module ISO_FORTRAN_ENV

1.3.157
vector subscript
section-subscript that is an array (6.5.3.3.2)

1.3.158
whole array
array component or array name without further qualification (6.5.2)

1.4 Notation, symbols and abbreviated terms

1.4.1 Syntax rules

1 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 (3.1) 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:

arg	for	argument	*attr*	for	attribute
decl	for	declaration	*def*	for	definition

desc	for	descriptor	*expr*	for	expression
int	for	integer	*op*	for	operator
spec	for	specifier	*stmt*	for	statement

- 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 2 and 3 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 1.9

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

If particular entities are substituted for *digit*, actual digit strings might be:

```
4
67
1999
10243852
```

1.4.2 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 (1.5). 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.

1.4.3 Assumed syntax rules

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

R101 *xyz-list* **is** *xyz* [, *xyz*] ...

R102 *xyz-name* **is** *name*

R103 *scalar-xyz* **is** *xyz*

C101 (R103) *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.

1.4.4 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* (R202). Expression hierarchy is described rigorously in the definition of *expr* (R722).

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 R515), 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*" (R406) and "*length-selector*" (R422).

1.4.5 Text conventions

1 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 1.10

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

1.5 Conformance

1 A program (2.2.2) 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 part of ISO/IEC 1539. A program unit (2.2.1) conforms to this part of ISO/IEC 1539 if it can be included in a program in a manner that allows the program to be standard conforming.

2 A processor conforms to this part of ISO/IEC 1539 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 an additional 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 (4.4);

(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 3;

(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 16;

(7) it contains the capability to detect and report the use within a submitted program unit of a nonstandard 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 nonstandard 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 part of ISO/IEC 1539 or the procedure has different requirements from those specified by this part of ISO/IEC 1539; and

(10) it contains the capability to detect and report the reason for rejecting a submitted program.

3 However, in a format specification that is not part of a FORMAT statement (10.2.1), a processor need not detect or report the use of deleted or obsolescent features, or the use of additional forms or relationships.

4 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 is given the EXTERNAL attribute (5.5.9) in its scope (2.2.1). A standard-conforming program shall not use nonstandard intrinsic procedures or modules that have been added by the processor.

5 Because a standard-conforming program may place demands on a processor that are not within the scope of this part of ISO/IEC 1539 or may include standard items that are not portable, such as external procedures defined by means other than Fortran, conformance to this part of ISO/IEC 1539 does not ensure that a program will execute consistently on all or any standard-conforming processors.

6 The semantics of facilities that are identified as processor dependent are not completely specified in this part of ISO/IEC 1539. They shall be provided, with methods or semantics determined by the processor.

7 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.

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

1.6 Compatibility

1.6.1 Previous Fortran standards

1 Table 1.3 lists the previous editions of the Fortran International Standard, along with their informal names.

Table 1.3: **Previous editions of the Fortran International Standard**

Official designation	Informal name
ISO R 1539-1972	FORTRAN 66
ISO 1539-1980	FORTRAN 77
ISO/IEC 1539:1991	Fortran 90
ISO/IEC 1539-1:1997	Fortran 95
ISO/IEC 1539-1:2004	Fortran 2003
ISO/IEC 1539-1:2010	Fortran 2008

1.6.2 New intrinsic procedures

1 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.

1.6.3 Fortran 2008 compatibility

1 Except as identified in this subclause, this part of ISO/IEC 1539 is an upward compatible 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 part of ISO/IEC 1539.

1.6.4 Fortran 2003 compatibility

1 Except as identified in this subclause, this part of ISO/IEC 1539 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 part of ISO/IEC 1539.

2 Fortran 2003 permitted a sequence type to have type parameters; that is not permitted by this part of ISO/IEC 1539.

3 Fortran 2003 specified that array constructors and structure constructors of finalizable type are finalized. This part of ISO/IEC 1539 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 required an explicit interface only for a procedure that was actually referenced in the scope, not merely passed as an actual argument. This part of ISO/IEC 1539 requires an explicit interface for a procedure under the conditions listed in 12.4.2.2, regardless of whether the procedure is referenced in the scope.

6 Fortran 2003 permitted the function result of a pure function to be a polymorphic allocatable variable, and to be finalizable by an impure final subroutine. These are not permitted by this part of ISO/IEC 1539.

7 Fortran 2003 permitted an INTENT (OUT) argument of a pure subroutine to be polymorphic; that is not permitted by this part of ISO/IEC 1539.

1.6.5 Fortran 95 compatibility

1 Except as identified in this subclause, this part of ISO/IEC 1539 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 part of ISO/IEC 1539.

2 Fortran 95 permitted defined assignment between character strings of the same rank and different kinds. This part of ISO/IEC 1539 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 part of ISO/IEC 1539.

- 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 part of ISO/IEC 1539.
- This part of ISO/IEC 1539 specifies a different output format for real zero values in list-directed and namelist output.
- If the processor can distinguish between positive and negative real zero, this part of ISO/IEC 1539 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 REAL(X) < 0 and negative zero imaginary part.
- This part of ISO/IEC 1539 has fewer restrictions on constant expressions than Fortran 95; this might affect whether a variable is considered to be automatic.
- The form produced by the G edit descriptor with *d* equal to zero differs from that specified by Fortran 95 for some values.

1.6.6 Fortran 90 compatibility

1 Except for the deleted features noted in Annex B.1, and except as identified in this subclause, this part of ISO/IEC 1539 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 part of ISO/IEC 1539.

2 The PAD= specifier in the INQUIRE statement in this part of ISO/IEC 1539 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 part of ISO/IEC 1539 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 part of ISO/IEC 1539 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 part of ISO/IEC 1539:

- 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 automatic);
- the G edit descriptor with *d* equal to zero for some values.

1.6.7 FORTRAN 77 compatibility

1 Except for the deleted features noted in Annex B.1, and except as identified in this subclause, this part of ISO/IEC 1539 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 part of ISO/IEC 1539. This part of ISO/IEC 1539 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 part of ISO/IEC 1539, yet remain a standard-conforming program. The following FORTRAN 77 features might have different interpretations in this part of ISO/IEC 1539.

- 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 part of ISO/IEC 1539 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 part of ISO/IEC 1539 specifies (5.6.7) 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 part of ISO/IEC 1539 specifies (9.6.4.5.3) 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 part of ISO/IEC 1539 specifies how rounding of values will affect the output field form, but FORTRAN 77 did not address this issue. Therefore, some FORTRAN 77 processors might produce an output form different from the output form produced by Fortran 2003 processors for certain combinations of values and G edit descriptors.
- If the processor can distinguish between positive and negative real zero, the behavior of the intrinsic function SIGN when the second argument is negative real zero is changed by this part of ISO/IEC 1539.

1.7 Deleted and obsolescent features

1.7.1 General

1 This part of ISO/IEC 1539 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 part of ISO/IEC 1539 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.

1.7.2 Nature of deleted features

1 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.

1.7.3 Nature of obsolescent features

1 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.

2 The obsolescent features are identified in the text of this part of ISO/IEC 1539 by a distinguishing type font (1.4.5).

3 A future revision of this part of ISO/IEC 1539 might delete an obsolescent feature if its use has become insignificant.

(Blank page)

2 Fortran concepts

2.1 High level syntax

1 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 2.1

Constraints and other information related to the rules that do not begin with R2 appear in the appropriate clause.

R201 *program* **is** *program-unit*
  [*program-unit*] ...

R202 *program-unit* **is** *main-program*
  or *external-subprogram*
  or *module*
  or *submodule*
  or *block-data*

R1101 *main-program* **is** [*program-stmt*]
  [*specification-part*]
  [*execution-part*]
  [*internal-subprogram-part*]
  end-program-stmt

R203 *external-subprogram* **is** *function-subprogram*
  or *subroutine-subprogram*

R1229 *function-subprogram* **is** *function-stmt*
  [*specification-part*]
  [*execution-part*]
  [*internal-subprogram-part*]
  end-function-stmt

R1235 *subroutine-subprogram* **is** *subroutine-stmt*
  [*specification-part*]
  [*execution-part*]
  [*internal-subprogram-part*]
  end-subroutine-stmt

R1104 *module* **is** *module-stmt*
  [*specification-part*]
  [*module-subprogram-part*]
  end-module-stmt

R1116 *submodule* **is** *submodule-stmt*
  [*specification-part*]
  [*module-subprogram-part*]
  end-submodule-stmt

R1120 *block-data* **is** *block-data-stmt*
  [*specification-part*]

end-block-data-stmt

R204 *specification-part* **is** [*use-stmt*] ...
[*import-stmt*] ...
[*implicit-part*]
[*declaration-construct*] ...

R205 *implicit-part* **is** [*implicit-part-stmt*] ...
implicit-stmt

R206 *implicit-part-stmt* **is** *implicit-stmt*
or *parameter-stmt*
or *format-stmt*
or *entry-stmt*

R207 *declaration-construct* **is** *derived-type-def*
or *enum-def*
or *format-stmt*
or *generic-stmt*
or *interface-block*
or *parameter-stmt*
or *procedure-declaration-stmt*
or *other-specification-stmt*
or *type-declaration-stmt*
or *entry-stmt*
or *stmt-function-stmt*

R208 *execution-part* **is** *executable-construct*
[*execution-part-construct*] ...

R209 *execution-part-construct* **is** *executable-construct*
or *format-stmt*
or *entry-stmt*
or *data-stmt*

R210 *internal-subprogram-part* **is** *contains-stmt*
[*internal-subprogram*] ...

R211 *internal-subprogram* **is** *function-subprogram*
or *subroutine-subprogram*

R1107 *module-subprogram-part* **is** *contains-stmt*
[*module-subprogram*] ...

R1108 *module-subprogram* **is** *function-subprogram*
or *subroutine-subprogram*
or *separate-module-subprogram*

R1239 *separate-module-subprogram* **is** *mp-subprogram-stmt*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-mp-subprogram-stmt

R212 *other-specification-stmt* **is** *access-stmt*
or *allocatable-stmt*
or *asynchronous-stmt*
or *bind-stmt*

or *codimension-stmt*
or *contiguous-stmt*
or *data-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*

R213 *executable-construct* **is** *action-stmt*
or *associate-construct*
or *block-construct*
or *case-construct*
or *critical-construct*
or *do-construct*
or *if-construct*
or *select-type-construct*
or *where-construct*
or *forall-construct*

R214 *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 *end-function-stmt*
or *end-mp-subprogram-stmt*
or *end-program-stmt*
or *end-subroutine-stmt*
or *endfile-stmt*
or *error-stop-stmt*
or *exit-stmt*
or *flush-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*
or *read-stmt*
or *return-stmt*
or *rewind-stmt*

or *stop-stmt*
or *sync-all-stmt*
or *sync-images-stmt*
or *sync-memory-stmt*
or *unlock-stmt*
or *wait-stmt*
or *where-stmt*
or *write-stmt*
or *computed-goto-stmt*
or *forall-stmt*

C201 (R208) An *execution-part* shall not contain an *end-function-stmt*, *end-mp-subprogram-stmt*, *end-program-stmt*, or *end-subroutine-stmt*.

2.2 Program unit concepts

2.2.1 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 are to 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 11 or 12.

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

NOTE 2.2

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.

2.2.2 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.

2.2.3 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 12.

2.2.4 Module

1 A module contains (or accesses from other modules) definitions that are to 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 11.

2.2.5 Submodule

1 A submodule extends a module or another submodule.

2 It may provide definitions (12.6) for procedures whose interfaces are declared (12.4.3.2) 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 11.

NOTE 2.3

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

2.3 Execution concepts

2.3.1 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.

2.3.2 Statement order

Table 2.1: **Requirements on statement ordering**

<table>
<tr><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE,
MODULE, SUBMODULE, or BLOCK DATA statement</td></tr>
<tr><td colspan="3">USE statements</td></tr>
<tr><td colspan="3">IMPORT statements</td></tr>
<tr><td rowspan="4">FORMAT
and
ENTRY
statements</td><td colspan="2">IMPLICIT NONE</td></tr>
<tr><td>PARAMETER
statements</td><td>IMPLICIT
statements</td></tr>
<tr><td>PARAMETER
and DATA
statements</td><td>Derived-type definitions,
interface blocks,
type declaration statements,
enumeration definitions,
procedure declarations,
specification statements,
and statement function statements</td></tr>
<tr><td>DATA
statements</td><td>Executable
constructs</td></tr>
<tr><td colspan="3">CONTAINS statement</td></tr>
<tr><td colspan="3">Internal subprograms
or module subprograms</td></tr>
<tr><td colspan="3">END statement</td></tr>
</table>

1 The syntax rules of subclause 2.1 specify the statement order within program units and subprograms. These rules are illustrated in Table 2.1 and Table 2.2. Table 2.1 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 2.2 shows which statements are allowed in some kinds of scoping units.

Table 2.2: **Statements allowed in scoping units**

	Kind of scoping unit						
Statement type	Main program	Module or submodule	Block data	External subprogram	Module subprogram	Internal subprogram	Interface body
USE	Yes	Yes	Yes	Yes	Yes	Yes	Yes
IMPORT	No	No	No	No	No	No	Yes
ENTRY	No	No	No	Yes	Yes	No	No
FORMAT	Yes	No	No	Yes	Yes	Yes	No
Misc. decl.s [1]	Yes	Yes	Yes	Yes	Yes	Yes	Yes
DATA	Yes	Yes	Yes	Yes	Yes	Yes	No
Derived-type	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Interface	Yes	Yes	No	Yes	Yes	Yes	Yes
Executable	Yes	No	No	Yes	Yes	Yes	No
CONTAINS	Yes	Yes	No	Yes	Yes	No	No
Statement function	Yes	No	No	Yes	Yes	Yes	No

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

2.3.3 The END statement

1 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 (8.2). Executing an *end-program-stmt* initiates normal termination of the image. 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*.

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

2.3.4 Program execution

1 Execution 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 (14.7), 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.

NOTE 2.4

Fortran control constructs (8.1, 8.2) control the progress of execution in each image. Image control statements (8.5.1) affect the relative progress of execution between images. Coarrays (2.4.7) provide a mechanism for accessing data on one image from another image.

NOTE 2.5

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 exact number of images is unlikely to be portable.

2.3.5 Execution sequence

1 Following the creation of a fixed number of instances of the program, 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 (8.5.2) executed by a single image are totally ordered, and segments executed by separate images are partially ordered by image control statements (8.5.1).

2 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 (8.2) changes the execution sequence. These statements explicitly specify a new starting place for the execution sequence.
- DO constructs, IF constructs, SELECT CASE constructs, and SELECT TYPE constructs contain an internal statement structure and execution of these constructs involves implicit internal transfer of control. See Clause 8 for the detailed semantics of each of these constructs.
- BLOCK constructs may contain specification expressions; see 8.1.4 for detailed semantics of this construct.
- END=, ERR=, and EOR= specifiers might result in a branch.
- Alternate returns might result in a branch.

2.3.6 Termination of execution

1 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.

2 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.

3 Error termination of execution of an image is initiated if an ERROR STOP statement is executed or as specified elsewhere in this part of ISO/IEC 1539. When error termination on an image has been initiated, the processor should initiate error termination on other images as quickly as possible.

4 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 2.6

As well as in the circumstances specified in this part of ISO/IEC 1539, error termination might be initiated by means other than Fortran.

NOTE 2.7

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

2.4 Data concepts

2.4.1 Type

1 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 4.1.

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

2.4.1.1 Intrinsic type

1 The intrinsic types are integer, real, complex, character, and logical. The properties of intrinsic types are described in 4.4.

2 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.

2.4.1.2 Derived type

1 Derived types may be parameterized. A scalar object of derived type is a structure; assignment of structures is defined intrinsically (7.2.1.3), but there are no intrinsic operations for structures. For each derived type, a structure constructor is available to create values (4.5.10). 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 (7.1.6).

2 Derived types are described further in 4.5.

2.4.2 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 4.4. 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.

2.4.3 Data entity

2.4.3.1 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.

2.4.3.2 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 (5.2) or implicitly (5.7).

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 6.

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).

2.4.3.2.1 Variable

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

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 2.8

A subobject of a local variable is also a local variable.

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

2.4.3.2.2 Constant

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

2 Named constants are defined using the PARAMETER attribute (5.5.13, 5.6.11). The syntax of literal constants is described in 4.4.

2.4.3.2.3 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 2.9

For example, given:

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

NOTE 2.9 (cont.)

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

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

2.4.3.3 Expression

1 An expression (7.1) 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 7.

2.4.3.4 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 (12.3.3). The value of a function result is determined by execution of the function.

2.4.4 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 16 describes the ways in which variables become defined and undefined and the association status of pointers becomes defined and undefined.

2.4.5 Reference

1 A data object is referenced when its value is required during execution. A procedure is referenced when it is executed.

2 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.

2.4.6 Array

1 An array may have up to fifteen dimensions, 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. The rank of a named array, once declared, is constant; the extents may be constant or may vary during execution.

2 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 (12.8), the element operations may be performed simultaneously or in any order.

3 A rank-one array may be constructed from scalars and other arrays and may be reshaped into any allowable array shape (4.8).

4 Arrays may be of any type and are described further in 6.5.

2.4.7 Coarray

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

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

3 The set of corresponding coarrays on all images 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 2.10

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.

4 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.

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

6 For a coindexed object, its cosubscript list determines the image index in the same way that a subscript list determines the subscript order value for an array element (6.5.3.2).

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

NOTE 2.11

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 2.12

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 (8.5.2) apply whether or not cosubscripts are used to access the coarray.

2.4.8 Pointer

1 A pointer has an association status which is either associated, disassociated, or undefined (16.5.2.2).

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.

2.4.9 Allocatable variables

1 The allocation status of an allocatable variable is either allocated or unallocated. An allocatable variable becomes allocated as described in 6.7.1.3. It becomes unallocated as described in 6.7.3.2.

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.

2.4.10 Storage

1 Many of the facilities of this part of ISO/IEC 1539 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 16.5.3.

2.5 Fundamental concepts

2.5.1 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.

2 A designator is used to identify a program constituent or a part thereof.

2.5.2 Statement keyword

1 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.

2.5.3 Other keywords

1 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.

2 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.

R215 *keyword* **is** *name*

NOTE 2.13

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.

2.5.4 Association

1 Association 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.

2 Also, storage association causes different entities to use the same storage.

2.5.5 Intrinsic

1 All intrinsic types, procedures, assignments, and operators may be used in any scoping unit without further definition or specification. Intrinsic modules (13.8, 14, 15.2) may be accessed by use association.

2.5.6 Operator

1 This part of ISO/IEC 1539 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 (4.5.5, 12.4.3.5).

2.5.7 Companion processors

1 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 (12.6.3), 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.

2 If a procedure is defined by means of a companion processor that is not the Fortran processor itself, this part of ISO/IEC 1539 refers to the C function that defines the procedure, although the procedure need not be defined by means of the C programming language.

NOTE 2.14

A companion processor might or might not be a mechanism that conforms to the requirements of ISO/IEC 9899:2011.

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.

(Blank page)

3 Lexical tokens and source form

3.1 Processor character set

3.1.1 Characters

1 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 (3.1.2), digits (3.1.3), underscore (3.1.4), special characters (3.1.5), and other characters (3.1.6).

2 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*.

R301 *alphanumeric-character* **is** *letter*
or *digit*
or *underscore*

3 Except for the currency symbol, the graphics used for the characters shall be as given in 3.1.2, 3.1.3, 3.1.4, and 3.1.5. However, the style of any graphic is not specified.

3.1.2 Letters

1 The twenty-six letters are:

2 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

3 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 (1.3).

NOTE 3.1

The following statements are equivalent:

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

3.1.3 Digits

1 The ten digits are:

2 0 1 2 3 4 5 6 7 8 9

3 The ten digits define the syntactic class *digit*.

3.1.4 Underscore

R302 *underscore* **is** _

3.1.5 Special characters

1 The special characters are shown in Table 3.1.

Table 3.1: **Special characters**

Character	Name of character	Character	Name of character
	Blank	;	Semicolon
=	Equals	!	Exclamation point
+	Plus	"	Quotation mark or quote
-	Minus	%	Percent
*	Asterisk	&	Ampersand
/	Slash	~	Tilde
\	Backslash	<	Less than
(	Left parenthesis	>	Greater than
)	Right parenthesis	?	Question mark
[	Left square bracket	'	Apostrophe
]	Right square bracket	`	Grave accent
{	Left curly bracket	^	Circumflex accent
}	Right curly bracket	|	Vertical line
,	Comma	$	Currency symbol
.	Decimal point or period	#	Number sign
:	Colon	@	Commercial at

2 Some of the special characters are used for operator symbols, bracketing, and various forms of separating and delimiting other lexical tokens.

3.1.6 Other characters

1 Additional characters may be representable in the processor, but may appear only in comments (3.3.2.3, 3.3.3.2), character constants (4.4.4), input/output records (9.2.2), and character string edit descriptors (10.3.2).

3.2 Low-level syntax

3.2.1 Tokens

1 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 %.

3.2.2 Names

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

R303 *name* **is** *letter* [*alphanumeric-character*] ...

C301 (R303) The maximum length of a *name* is 63 characters.

NOTE 3.2

Examples of names:

```
A1
NAME_LENGTH          (single underscore)
S_P_R_E_A_D__O_U_T   (two consecutive underscores)
TRAILER_             (trailing underscore)
```

NOTE 3.3

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.

3.2.3 Constants

R304	*constant*	**is**	*literal-constant*
		or	*named-constant*
R305	*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*
R306	*named-constant*	**is**	*name*
R307	*int-constant*	**is**	*constant*

C302 (R307) *int-constant* shall be of type integer.

3.2.4 Operators

R308	*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*
R707	*power-op*	**is**	**
R708	*mult-op*	**is**	*
		or	/
R709	*add-op*	**is**	+
		or	–
R711	*concat-op*	**is**	//
R713	*rel-op*	**is**	.EQ.
		or	.NE.
		or	.LT.
		or	.LE.
		or	.GT.
		or	.GE.
		or	==
		or	/=
		or	<
		or	<=
		or	>
		or	>=

R718 *not-op* **is** .NOT.

R719 *and-op* **is** .AND.

R720 *or-op* **is** .OR.

R721 *equiv-op* **is** .EQV.
or .NEQV.

R309 *defined-operator* **is** *defined-unary-op*
or *defined-binary-op*
or *extended-intrinsic-op*

R703 *defined-unary-op* **is** . *letter* [*letter*] ... .

R723 *defined-binary-op* **is** . *letter* [*letter*] ... .

R310 *extended-intrinsic-op* **is** *intrinsic-operator*

3.2.5 Statement labels

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

R311 *label* **is** *digit* [*digit* [*digit* [*digit* [*digit*]]]]

C303 (R311) At least one digit in a *label* shall be nonzero.

2 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 3.4

For example:

```
99999
10
 010
```

are all statement labels. The last two are equivalent.

3 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 (8.2) is used to identify that statement as the possible destination of a branch.
- The label on a FORMAT statement (10.2.1) is used to identify that statement as the format specification for a data transfer statement (9.6).
- In some forms of the DO construct (8.1.6), the terminal statement of the construct is identified by a label.

3.2.6 Delimiters

1 A lexical token that is a delimiter is a (,), /, [,], (/, or /).

3.3 Source form

3.3.1 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 3.3.2 applies only to free form source. Subclause 3.3.3 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.

3.3.2 Free source form

3.3.2.1 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 (4.4.4), 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.

3.3.2.2 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 3.5

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

```
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 3.2: **Adjacent keywords where separating blanks are optional**

BLOCK DATA	END ENUM	END SELECT
DOUBLE PRECISION	END FILE	END SUBMODULE
ELSE IF	END FORALL	END SUBROUTINE
ELSE WHERE	END FUNCTION	END TYPE
END ASSOCIATE	END IF	END WHERE
END BLOCK	END INTERFACE	GO TO
END BLOCK DATA	END MODULE	IN OUT
END CRITICAL	END PROCEDURE	SELECT CASE
END DO	END PROGRAM	SELECT TYPE

3.3.2.3 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 may follow the last statement of a program unit. Comments have no effect on the interpretation of the program unit.

NOTE 3.6

This part of ISO/IEC 1539 does not restrict the number of consecutive comment lines.

3.3.2.4 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 and shall not be followed by commentary. 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 "&".

3.3.2.5 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.

3.3.2.6 Free form statements

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

NOTE 3.7

No Fortran statement begins with a digit.

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

3.3.3 Fixed source form

3.3.3.1 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.

3.3.3.2 Fixed form commentary

1 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 may follow the last statement of a program unit. Comments have no effect on the interpretation of the program unit.

NOTE 3.8

This part of ISO/IEC 1539 does not restrict the number of consecutive comment lines.

3.3.3.3 Fixed form statement continuation

1 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 3.9

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.

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

3.3.3.4 Fixed 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, 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.

3.3.3.5 Fixed form statements

1 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.

3.4 Including source text

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

2 INCLUDE *char-literal-constant*

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

4 An INCLUDE line is not a Fortran statement.

5 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.

6 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.

7 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.

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

NOTE 3.10

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.

4 Types

4.1 Characteristics of types

4.1.1 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.

4.1.2 Type classification

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

2 This part of ISO/IEC 1539 defines five intrinsic types: integer, real, complex, character, and logical.

3 A derived type is one that is defined by a derived-type definition (4.5.2) or by an intrinsic module. It shall be used only where it is accessible (4.5.2.2). An intrinsic type is always accessible.

4.1.3 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 part of ISO/IEC 1539. 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 individual components. The set of valid values for a derived type is as defined in 4.5.8.

4.1.4 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 4.4.

3 A structure constructor (4.5.10) that is a constant expression (7.1.12) denotes a scalar constant value of derived type. An array constructor (4.8) that is a constant expression denotes a constant array value of intrinsic or derived type.

4 A constant value can be named (5.5.13, 5.6.11).

4.1.5 Operations

1 For each of the intrinsic types, a set of operations and corresponding operators is defined intrinsically. These are described in Clause 7. The intrinsic set can be augmented with operations and operators defined by functions with the OPERATOR interface (12.4.3.2). Operator definitions are described in Clauses 7 and 12.

2 For derived types, there are no intrinsic operations. Operations on derived types can be defined by the program (4.5.11).

4.2 Type parameters

1 A type might be parameterized. In this case, the set of values, the syntax for denoting the values, and the set of operations on the values of the type depend on the values of the parameters.

2 The intrinsic types are all parameterized. Derived types may be defined to be parameterized.

3 A type parameter is either a kind type parameter or a length type parameter. All type parameters are of type integer.

4 A kind type parameter may be used in constant and specification expressions within the derived-type definition for the type (4.5.4); it participates in generic resolution (12.5.5.2). Each of the intrinsic types has a kind type parameter named KIND, which is used to distinguish multiple representations of the intrinsic type.

NOTE 4.1

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.

A type parameter of a derived type can be specified to be a kind type parameter in order to allow generic resolution based on the parameter; that is to allow a single generic to include two specific procedures that have interfaces distinguished only by the value of a kind type parameter of a dummy argument. All generic references are resolvable at compile time.

5 A length type parameter may be used in specification expressions within the derived-type definition for the type, but it shall not be used in constant expressions. The intrinsic character type has a length type parameter named LEN, which is the length of the string.

NOTE 4.2

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.

6 A type parameter value may be specified by a type specification (4.4, 4.5.9).

R401	*type-param-value*	**is**	*scalar-int-expr*
		or	*
		or	:

C401 (R401) The *type-param-value* for a kind type parameter shall be a constant expression.

C402 (R401) A colon shall not be used as a *type-param-value* except in the declaration of an entity or component that has the POINTER or ALLOCATABLE attribute.

7 A colon as a *type-param-value* specifies a deferred type parameter.

8 The values of the deferred type parameters of an object are determined by successful execution of an ALLOCATE statement (6.7.1), execution of an intrinsic assignment statement (7.2.1.3), execution of a pointer assignment statement (7.2.2), or by argument association (12.5.2).

NOTE 4.3

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.

9 An asterisk as a *type-param-value* specifies that a length type parameter is an assumed type parameter. It is used for a dummy argument to assume the type parameter value from the effective argument, for an associate name in a SELECT TYPE construct to assume the type parameter value from the corresponding selector, and for a named constant of type character to assume the character length from the *constant-expr*.

4.3 Types, type specifiers, and values

4.3.1 Relationship of types and values to objects

1 The name of a type serves as a type specifier and may be used to declare objects of that type. A declaration 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 5 describes the way in which a data object is declared and how its type and other attributes are specified.

2 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.

3 A variable is a data object. The type and type parameters of a variable determine which values that variable may take. Assignment (7.2) provides one means of defining or redefining the value of a variable of any type.

4 The type of a variable determines the operations that may be used to manipulate the variable.

4.3.2 Type specifiers and type compatibility

4.3.2.1 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*.

R402 *type-spec* **is** *intrinsic-type-spec*
or *derived-type-spec*

C403 (R402) The *derived-type-spec* shall not specify an abstract type (4.5.7).

R403 *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 (*)

C404 (R403) In a *declaration-type-spec*, every *type-param-value* that is not a colon or an asterisk shall be a *specification-expr*.

C405 (R403) In a *declaration-type-spec* that uses the CLASS keyword, *derived-type-spec* shall specify an extensible type (4.5.7).

C406 (R403) TYPE(*derived-type-spec*) shall not specify an abstract type (4.5.7).

C407 (R402) In TYPE(*intrinsic-type-spec*) the *intrinsic-type-spec* shall not end with a comma.

C408 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 4.4

A *type-spec* is used in an array constructor, a SELECT TYPE construct, or an ALLOCATE statement. Elsewhere, a *declaration-type-spec* is used.

4.3.2.2 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 Where a data entity is declared explicitly using the TYPE type specifier to be of derived type, the specified derived type shall have been defined previously in the scoping unit or be accessible there by use or host association. 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.

3 An entity that is declared using the TYPE(*) type specifier is assumed-type and is an unlimited polymorphic entity. Its dynamic type and type parameters are assumed from its effective argument.

C409 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.

C410 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.

C411 An assumed-type actual argument that corresponds to an assumed-rank dummy argument shall be assumed-shape or assumed-rank.

4.3.2.3 CLASS type specifier

1 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.

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

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

4 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 4.5

Given

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

NOTE 4.5 (cont.)

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

5 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.

6 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 (8.1.3) 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.

4.4 Intrinsic types

4.4.1 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.

R404	*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*]
R405	*integer-type-spec*	**is**	INTEGER [*kind-selector*]
R406	*kind-selector*	**is**	([KIND =] *scalar-int-constant-expr*)

C412 (R406) The value of *scalar-int-constant-expr* shall be nonnegative and shall specify a representation method that exists on the processor.

4.4.2 Intrinsic operations on intrinsic types

1 Intrinsic numeric operations are defined as specified in 7.1.5.2.1 for the numeric intrinsic types. Relational intrinsic operations are defined as specified in 7.1.5.5 for numeric and character intrinsic types. The intrinsic concatenation operation is defined as specified in 7.1.5.3 for the character type. Logical intrinsic operations are defined as specified in 7.1.5.4 for the logical type.

4.4.3 Numeric intrinsic types

4.4.3.1 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 (13.7.90). The decimal exponent range of a representation method is returned by the intrinsic function RANGE (13.7.140). The intrinsic function SELECTED_INT_KIND (13.7.151) 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.

4 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.

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

R407	*signed-int-literal-constant*	**is**	[*sign*] *int-literal-constant*
R408	*int-literal-constant*	**is**	*digit-string* [_ *kind-param*]
R409	*kind-param*	**is**	*digit-string*
		or	*scalar-int-constant-name*
R410	*signed-digit-string*	**is**	[*sign*] *digit-string*
R411	*digit-string*	**is**	*digit* [*digit*] ...
R412	*sign*	**is**	+
		or	−

C413 (R409) A *scalar-int-constant-name* shall be a named constant of type integer.

C414 (R409) The value of *kind-param* shall be nonnegative.

C415 (R408) The value of *kind-param* shall specify a representation method that exists on the processor.

6 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.

7 An integer constant is interpreted as a decimal value.

NOTE 4.6

Examples of signed integer literal constants are:

```
473
+56
-101
21_2
21_SHORT
1976354279568241_8
```

where SHORT is a scalar integer named constant.

4.4.3.2 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 (13.7.90).

2 The decimal precision, decimal exponent range, and radix of an approximation method are returned by the intrinsic functions PRECISION (13.7.133), RADIX (13.7.136) and RANGE (13.7.140). The intrinsic function

SELECTED_REAL_KIND (13.7.152) returns a kind value based on specified precision, range, and radix requirements.

NOTE 4.7

See C.1.1 for remarks concerning selection of approximation methods.

3 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 4.8

On a processor that distinguishes between 0.0 and −0.0,

```
( X >= 0.0 )
```

evaluates to true if X = 0.0 or if X = −0.0,

```
( X < 0.0 )
```

evaluates to false for X = −0.0, and

```
IF (X) 1,2,3
```

causes a transfer of control to the branch target statement with the statement label "2" for both X = 0.0 and 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.

4 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.

5 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.

6 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.

R413	*signed-real-literal-constant*	**is**	[*sign*] *real-literal-constant*
R414	*real-literal-constant*	**is**	*significand* [*exponent-letter exponent*] [_ *kind-param*]
		or	*digit-string exponent-letter exponent* [_ *kind-param*]
R415	*significand*	**is**	*digit-string* . [*digit-string*]
		or	. *digit-string*
R416	*exponent-letter*	**is**	E
		or	D
R417	*exponent*	**is**	*signed-digit-string*

C416 (R414) If both *kind-param* and *exponent-letter* appear, *exponent-letter* shall be E.

C417 (R414) The value of *kind-param* shall specify an approximation method that exists on the processor.

7 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.

8 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.

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

NOTE 4.9

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.

4.4.3.3 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 (13.7.90).

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.

R418	*complex-literal-constant*	**is**	(*real-part* , *imag-part*)
R419	*real-part*	**is**	*signed-int-literal-constant*
		or	*signed-real-literal-constant*
		or	*named-constant*
R420	*imag-part*	**is**	*signed-int-literal-constant*
		or	*signed-real-literal-constant*
		or	*named-constant*

C418 (R418) 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 4.10

Examples of complex literal constants are:

```
(1.0, -1.0)
(3, 3.1E6)
(4.0_4, 3.6E7_8)
( 0., PI)         ! where PI is a previously declared named real constant.
```

4.4.4 Character type

4.4.4.1 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 (13.7.90). The intrinsic function SELECTED_CHAR_KIND (13.7.150) 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.

4.4.4.2 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 (3.1). By supplying nondefault character kinds, the processor may support additional character sets. The characters available in nondefault character kinds are not specified by this part of ISO/IEC 1539, except that one character in each nondefault character set shall be designated as a blank character to be used as a padding character.

R421	*char-selector*	**is**	*length-selector*
		or	(LEN = *type-param-value* , ■
			■ KIND = *scalar-int-constant-expr*)
		or	(*type-param-value* , ■
			■ [KIND =] *scalar-int-constant-expr*)
		or	(KIND = *scalar-int-constant-expr* ■
			■ [, LEN =*type-param-value*])
R422	*length-selector*	**is**	([LEN =] *type-param-value*)
		or	* *char-length* [,]

R423 *char-length* **is** (*type-param-value*)
or *int-literal-constant*

C419 (R421) The value of *scalar-int-constant-expr* shall be nonnegative and shall specify a representation method that exists on the processor.

C420 (R423) The *int-literal-constant* shall not include a *kind-param*.

C421 (R423) A *type-param-value* in a *char-length* shall be a colon, asterisk, or *specification-expr*.

C422 (R421 R422 R423) 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 (8.1.9), or
- in an external function, to declare the character length parameter of the function result.

C423 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.

C424 A function name declared with an asterisk *type-param-value* shall not be an array, a pointer, elemental, recursive, or pure.

C425 (R422) The optional comma in a *length-selector* is permitted only in a *declaration-type-spec* in a *type-declaration-stmt*.

C426 (R422) The optional comma in a *length-selector* is permitted only if no double-colon separator appears in the *type-declaration-stmt*.

C427 (R421) 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 (4.2). 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 the length of the 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 the 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.

4.4.4.3 Character literal constant

1 The syntax of a character literal constant is given by R424.

R424 *char-literal-constant* **is** [*kind-param* _] ' [*rep-char*] ... '
or [*kind-param* _] " [*rep-char*] ... "

C428 (R424) 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.

3 For the type character with kind *kind-param*, if it appears, and for default character otherwise, a representable character, *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.

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

5 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.

6 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.

NOTE 4.11

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.

NOTE 4.12

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".

4.4.4.4 Collating sequence

1 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 CHAR (13.7.35) and ICHAR (13.7.78) provide conversions between the characters and the integers according to this mapping.

NOTE 4.13

For example:

NOTE 4.13 (cont.)

```
ICHAR ( 'X' )
```

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

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

- ICHAR ('A') < ICHAR ('B') < ... < 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 part of ISO/IEC 1539. The collating sequence for the ISO 10646 character kind is as specified in ISO/IEC 10646.

NOTE 4.14

The intrinsic functions ACHAR (13.7.3) and IACHAR (13.7.71) provide conversions between characters and corresponding integer values according to the ASCII collating sequence.

5 The intrinsic functions LGT, LGE, LLE, and LLT (13.7.96-13.7.99) 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 (3.1).

4.4.5 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 (13.7.90).

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.).

R425	*logical-literal-constant*	**is**	.TRUE. [_ *kind-param*]
		or	.FALSE. [_ *kind-param*]

C429 (R425) The value of *kind-param* shall specify a representation method that exists on the processor.

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.

4.5 Derived types

4.5.1 Derived type concepts

1 Additional types may 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 may be parameterized by multiple type parameters, each of which is defined to be either a kind or length type parameter and may 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.

4 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.

5 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 (6.4.2).

6 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.

7 By default, no storage sequence is implied by the order of the component definitions. However, a storage order is implied for a sequence type (4.5.2.3). If the derived type has the BIND attribute, the storage sequence is that required by the companion processor (2.5.7, 15.3.4).

8 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 4.15

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

4.5.2 Derived-type definition

4.5.2.1 Syntax

R426 *derived-type-def* **is** *derived-type-stmt*
[*type-param-def-stmt*] ...
[*private-or-sequence*] ...
[*component-part*]
[*type-bound-procedure-part*]
end-type-stmt

R427 *derived-type-stmt* **is** TYPE [[, *type-attr-spec-list*] ::] *type-name* ■
■ [(*type-param-name-list*)]

R428 *type-attr-spec* **is** ABSTRACT
or *access-spec*
or BIND (C)
or EXTENDS (*parent-type-name*)

C430 (R427) A derived type *type-name* shall not be DOUBLEPRECISION or the same as the name of any intrinsic type defined in this part of ISO/IEC 1539.

C431 (R427) The same *type-attr-spec* shall not appear more than once in a given *derived-type-stmt*.

C432 (R427) The same *type-param-name* shall not appear more than once in a *derived-type-stmt*.

C433 (R428) A *parent-type-name* shall be the name of a previously defined extensible type (4.5.7).

C434 (R426) If the type definition contains or inherits (4.5.7.2) a deferred type-bound procedure (4.5.5), ABSTRACT shall appear.

C435 (R426) If ABSTRACT appears, the type shall be extensible.

C436 (R426) If EXTENDS appears, SEQUENCE shall not appear.

C437 (R426) If EXTENDS appears and the type being defined has a coarray ultimate component, its parent type shall have a coarray ultimate component.

C438 (R426) If EXTENDS appears and the type being defined has a potential subobject component of type LOCK_TYPE from the intrinsic module ISO_FORTRAN_ENV, its parent type shall be LOCK_TYPE or have a potential subobject component of type LOCK_TYPE.

R429 *private-or-sequence* **is** *private-components-stmt*
or *sequence-stmt*

C439 (R426) The same *private-or-sequence* shall not appear more than once in a given *derived-type-def*.

R430 *end-type-stmt* **is** END TYPE [*type-name*]

C440 (R430) If END TYPE is followed by a *type-name*, the *type-name* shall be the same as that in the corresponding *derived-type-stmt*.

1 Derived types with the BIND attribute are subject to additional constraints as specified in 15.3.4.

NOTE 4.16

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
```

4.5.2.2 Accessibility

1 The accessibility of a type name is determined as specified in 5.5.2. The accessibility of a type name does not affect, and is not affected by, the accessibility of its components and type-bound procedures.

2 If a type definition is private, then the type name, and thus the structure constructor (4.5.10) for the type, are accessible only within the module containing the definition, and within its descendants.

NOTE 4.17

An example of a type with a private name is:

```
TYPE, PRIVATE :: AUXILIARY
   LOGICAL :: DIAGNOSTIC
   CHARACTER (LEN = 20) :: MESSAGE
END TYPE AUXILIARY
```

Such a type would be accessible only within the module in which it is defined, and within its descendants.

4.5.2.3 Sequence type

R431 *sequence-stmt* **is** SEQUENCE

C441 (R426) 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 4.18

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 4.19

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.

4.5.2.4 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 4.20

An example of declaring two entities with reference to the same derived-type definition is:

```
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 referenced the module.

NOTE 4.21

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

```
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

NOTE 4.21 (cont.)

actual argument PROGRAMMER would not be of the same type as the dummy argument POSITION. Thus, the program would not be standard-conforming.

NOTE 4.22

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

4.5.3 Derived-type parameters

4.5.3.1 Type parameter definition statement

R432 *type-param-def-stmt* **is** *integer-type-spec*, *type-param-attr-spec* :: ■
■ *type-param-decl-list*

R433 *type-param-decl* **is** *type-param-name* [= *scalar-int-constant-expr*]

C442 (R432) 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*.

C443 (R432) 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*.

R434 *type-param-attr-spec* **is** KIND
or LEN

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 (7.2.1.3) 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 (7.1.11) in the *derived-type-def*. A kind type parameter may also be used as a primary in a constant expression (7.1.12) in the *derived-type-def*.

NOTE 4.23

The following example uses derived-type parameters.

```
TYPE humongous_matrix(k, d)
  INTEGER, KIND :: k = kind(0.0)
  INTEGER(selected_int_kind(12)), LEN :: d
    !-- Specify a 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`.

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

NOTE 4.23 (cont.)

```
    END TYPE
```

4.5.3.2 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.

2 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 (4.5.7) 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 4.24

Given

```
    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`.

4.5.4 Components

4.5.4.1 Component definition statement

R435	*component-part*	**is**	[*component-def-stmt*] ...
R436	*component-def-stmt*	**is**	*data-component-def-stmt*
		or	*proc-component-def-stmt*
R437	*data-component-def-stmt*	**is**	*declaration-type-spec* [[, *component-attr-spec-list*] ::] ■
			■ *component-decl-list*
R438	*component-attr-spec*	**is**	*access-spec*
		or	ALLOCATABLE
		or	CODIMENSION *lbracket coarray-spec rbracket*
		or	CONTIGUOUS
		or	DIMENSION (*component-array-spec*)
		or	POINTER
R439	*component-decl*	**is**	*component-name* [(*component-array-spec*)] ■
			■ [*lbracket coarray-spec rbracket*] ■
			■ [* *char-length*] [*component-initialization*]
R440	*component-array-spec*	**is**	*explicit-shape-spec-list*
		or	*deferred-shape-spec-list*

C444 (R437) No *component-attr-spec* shall appear more than once in a given *component-def-stmt*.

C445 (R437) 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.

C446 (R437) If the POINTER or ALLOCATABLE attribute is specified, each *component-array-spec* shall be a *deferred-shape-spec-list*.

C447 (R437) If a *coarray-spec* appears, it shall be a *deferred-coshape-spec-list* and the component shall have the ALLOCATABLE attribute.

C448 (R437) If a *coarray-spec* appears, the component shall not be of type C_PTR or C_FUNPTR (15.3.3).

C449 A data component whose type has a coarray ultimate component shall be a nonpointer nonallocatable scalar and shall not be a coarray.

C450 (R437) If neither the POINTER nor the ALLOCATABLE attribute is specified, each *component-array-spec* shall be an *explicit-shape-spec-list*.

C451 (R440) 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.

C452 (R437) A component shall not have both the ALLOCATABLE and POINTER attributes.

C453 (R437) If the CONTIGUOUS attribute is specified, the component shall be an array with the POINTER attribute.

C454 (R439) The * *char-length* option is permitted only if the component is of type character.

C455 (R436) 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 4.25

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*.

R441	*proc-component-def-stmt*	**is**	PROCEDURE ([*proc-interface*]) , ■ ■ *proc-component-attr-spec-list* :: *proc-decl-list*

NOTE 4.26

See 12.4.3.7 for definitions of *proc-interface* and *proc-decl*.

R442	*proc-component-attr-spec*	**is**	POINTER
		or	PASS [(*arg-name*)]
		or	NOPASS
		or	*access-spec*

C456 (R441) The same *proc-component-attr-spec* shall not appear more than once in a given *proc-component-def-stmt*.

C457 (R441) POINTER shall appear in each *proc-component-attr-spec-list*.

C458 (R441) If the procedure pointer component has an implicit interface or has no arguments, NOPASS shall be specified.

C459 (R441) If PASS (*arg-name*) appears, the interface of the procedure pointer component shall have a dummy argument named *arg-name*.

C460 (R441) PASS and NOPASS shall not both appear in the same *proc-component-attr-spec-list*.

1 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*.

4.5.4.2 Array components

1 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 (5.5.8.2), 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 4.27

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

```
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:

```
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 4.28

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

```
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 4.29

Default initialization of an explicit-shape array component can be specified by a constant expression consisting of an array constructor (4.8), or of a single scalar that becomes the value of each array element.

4.5.4.3 Coarray components

1 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 4.30

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

```
TYPE GRID_TYPE
   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.

4.5.4.4 Pointer components

1 A component is a pointer (2.4.8) if its *component-attr-spec-list* contains the POINTER attribute. A pointer component may be a data pointer or a procedure pointer.

NOTE 4.31

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

```
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 (6.7.1) or the pointer component could be associated with a target by a pointer assignment statement (7.2.2).

4.5.4.5 The passed-object dummy argument

1 A passed-object dummy argument is a distinguished dummy argument of a procedure pointer component or type-bound procedure. It affects procedure overriding (4.5.7.3) and argument association (12.5.2.2).

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

3 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.

4 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.

C461 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 (4.3.2.3) if and only if the type being defined is extensible (4.5.7). It shall not have the VALUE attribute.

NOTE 4.32

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.

4.5.4.6 Default initialization for components

1 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.

2 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.

R443	*component-initialization*	**is**	= *constant-expr*
		or	=> *null-init*
		or	=> *initial-data-target*
R444	*initial-data-target*	**is**	*designator*

C462 (R437) If *component-initialization* appears, a double-colon separator shall appear before the *component-decl-list*.

C463 (R437) If *component-initialization* appears, every type parameter and array bound of the component shall be a colon or constant expression.

C464 (R437) 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*.

C465 (R443) If *initial-data-target* appears, *component-name* shall be data-pointer-initialization compatible with it.

C466 (R444) The *designator* shall designate a nonallocatable 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 (1.3) or becomes disassociated as specified in 16.5.2.4.

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 16.5.2.3.

5 If *initial-proc-target* (12.4.3.7) 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 16.5.2.3.

6 If *constant-expr* appears for a nonpointer component, that component in any object of the type is initially defined (16.6.3) or becomes defined as specified in 16.6.5 with the value determined from *constant-expr*. If necessary, the value is converted according to the rules of intrinsic assignment (7.2.1.3) 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 4.34). Explicit initialization in a type declaration statement (5.2) overrides default initialization (see Note 4.33). Unlike explicit initialization, default initialization does not imply that the object has the SAVE attribute.

7 A subcomponent (6.4.2) 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 4.33

It is not required that initialization be specified for each component of a derived type. For example:

```
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 4.34

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 4.33:

```
TYPE SINGLE_SCORE
    TYPE(DATE) :: PLAY_DAY = TODAY
    INTEGER 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 4.35

Arrays of structures can be declared with elements that are partially or totally initialized by default. Continuing the example of Note 4.34 :

```
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 4.36

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:

NOTE 4.36 (cont.)

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

A type such as this can be used to construct linked lists of objects of type NODE. See C.1.5 for an example. Linked lists can also be constructed using allocatable components.

NOTE 4.37

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

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

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

4.5.4.7 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 (4.5.7).

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 4.38

Given the same type definitions as in Note 4.24, 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.

4.5.4.8 Component accessibility

R445 *private-components-stmt* **is** PRIVATE

C467 (R445) A *private-components-stmt* is permitted only if the type definition is within the specification part of a module.

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 4.39

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

NOTE 4.40

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 4.41

An example of a type with private components is:

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

Such a type definition is accessible in any scoping unit accessing the module via a USE statement; however, the components X and Y are accessible only within the module, and within its descendants.

NOTE 4.42

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

```
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.

4.5.5 Type-bound procedures

R446 *type-bound-procedure-part* **is** *contains-stmt*
  [*binding-private-stmt*]
  [*type-bound-proc-binding*] ...

R447 *binding-private-stmt* **is** PRIVATE

C468 (R446) A *binding-private-stmt* is permitted only if the type definition is within the specification part of a module.

R448 *type-bound-proc-binding* **is** *type-bound-procedure-stmt*
  or *type-bound-generic-stmt*
  or *final-procedure-stmt*

R449 *type-bound-procedure-stmt* **is** PROCEDURE [[, *binding-attr-list*] ::] *type-bound-proc-decl-list*
  or PROCEDURE (*interface-name*), *binding-attr-list* :: *binding-name-list*

R450 *type-bound-proc-decl* **is** *binding-name* [=> *procedure-name*]

C469 (R449) If => *procedure-name* appears in a *type-bound-proc-decl*, the double-colon separator shall appear.

C470 (R449) The *procedure-name* shall be the name of an accessible module procedure or an external procedure that has an explicit interface.

1 If neither => *procedure-name* nor *interface-name* appears in a *type-bound-proc-decl*, it is as though => *procedure-name* had appeared with a procedure name the same as the binding name.

R451 *type-bound-generic-stmt* **is** GENERIC [, *access-spec*] :: *generic-spec* => *binding-name-list*

C471 (R451) 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.

C472 (R451) Each *binding-name* in *binding-name-list* shall be the name of a specific binding of the type.

C473 (R451) If *generic-spec* is not *generic-name*, each of its specific bindings shall have a passed-object dummy argument (4.5.4.5).

C474 (R451) If *generic-spec* is OPERATOR (*defined-operator*), the interface of each binding shall be as specified in 12.4.3.5.2.

C475 (R451) If *generic-spec* is ASSIGNMENT (=), the interface of each binding shall be as specified in 12.4.3.5.3.

C476 (R451) If *generic-spec* is *defined-io-generic-spec*, the interface of each binding shall be as specified in 9.6.4.8. The type of the `dtv` argument shall be *type-name*.

R452 *binding-attr* **is** PASS [(*arg-name*)]
or NOPASS
or NON_OVERRIDABLE
or DEFERRED
or *access-spec*

C477 (R452) The same *binding-attr* shall not appear more than once in a given *binding-attr-list*.

C478 (R449) If the interface of the binding has no dummy argument of the type being defined, NOPASS shall appear.

C479 (R449) If PASS (*arg-name*) appears, the interface of the binding shall have a dummy argument named *arg-name*.

C480 (R452) PASS and NOPASS shall not both appear in the same *binding-attr-list*.

C481 (R452) NON_OVERRIDABLE and DEFERRED shall not both appear in the same *binding-attr-list*.

C482 (R452) DEFERRED shall appear if and only if *interface-name* appears.

C483 (R449) An overriding binding (4.5.7.3) shall have the DEFERRED attribute only if the binding it overrides is deferred.

C484 (R449) A binding shall not override an inherited binding (4.5.7.2) 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 (4.5.4.5). 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.

5 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.

6 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 4.43

An example of a type and a type-bound procedure is:

```
TYPE POINT
  REAL :: X, Y
CONTAINS
  PROCEDURE, PASS :: LENGTH => POINT_LENGTH
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
```

7 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 4.44

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 (16.3).

8 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.

9 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 4.45

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*.

4.5.6 Final subroutines

4.5.6.1 FINAL statement

R453 *final-procedure-stmt* **is** FINAL [::] *final-subroutine-name-list*

C485 (R453) 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.

C486 (R453) A *final-subroutine-name* shall not be one previously specified as a final subroutine for that type.

C487 (R453) A final subroutine shall not have a dummy argument with the same kind type parameters and rank as the dummy argument of another final subroutine of that type.

1 The FINAL statement specifies that each procedure it names is a final subroutine. A final subroutine might be executed when a data entity of that type is finalized (4.5.6.2).

2 A derived type is finalizable if and only if it has a final subroutine or a nonpointer, nonallocatable component of finalizable type. A nonpointer data entity is finalizable if and only if it is of finalizable type. No other entity is finalizable.

NOTE 4.46

Final subroutines are effectively always "accessible". They are called for entity finalization regardless of the accessibility of the type, its other type-bound procedures, or the subroutine name itself.

NOTE 4.47

Final subroutines are not inherited through type extension and cannot be overridden. The final subroutines of the parent type are called after any additional final subroutines of an extended type are called.

4.5.6.2 The finalization process

1 Only finalizable entities are finalized. When an entity is finalized, the following steps are carried out in sequence.

(1) If the dynamic type of the entity has a 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 actual argument. Otherwise, if there is an elemental final subroutine whose dummy argument has the same kind type parameters as the entity being finalized, it is called with the entity as an actual argument. Otherwise, no subroutine is called at this point.

(2) All finalizable components that appear in the type definition are finalized in a processor-dependent order. If the entity being finalized is an array, each finalizable component of each element of that entity is finalized separately.

(3) If the entity is of extended type and the parent type is finalizable, the parent component is finalized.

2 If several entities are to be finalized as a consequence of an event specified in 4.5.6.3, the order in which they are finalized is processor dependent. During this process, execution of a final subroutine for one of these entities shall not reference or define any of the other entities that have already been finalized.

3 If an object is not finalized, it retains its definition status and does not become undefined.

NOTE 4.48

An implementation might need to ensure that when an event causes more than one coarray to be deallocated, they are deallocated in the same order on all images.

4.5.6.3 When finalization occurs

1 When an intrinsic assignment statement is executed (7.2.1.3), if the variable is not an unallocated allocatable variable, it is finalized after evaluation of *expr* and before the definition of the variable. If the variable is an allocated allocatable variable that would be deallocated by intrinsic assignment, the finalization occurs before the deallocation.

2 When a pointer is deallocated its target is finalized. When an allocatable entity is deallocated, it is finalized unless it is the variable in an intrinsic assignment statement or a component thereof. If an error condition occurs during deallocation, it is processor dependent whether finalization occurs.

3 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 (16.6.6, item (3)).

4 A nonpointer nonallocatable local variable of a BLOCK construct is finalized immediately before it would become undefined due to termination of the BLOCK construct (16.6.6, item (22)).

5 If an executable construct references a nonpointer function, the result is finalized after execution of the innermost executable construct containing the reference.

6 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.

7 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.

8 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 4.49

If finalization is used for storage management, it often needs to be combined with defined assignment.

4.5.6.4 Entities that are not finalized

1 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 4.50

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.

4.5.7 Type extension

4.5.7.1 Extensible, extended, and abstract types

1 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.

2 A type with the EXTENDS attribute is an extended type; its parent type is the type named in the EXTENDS *type-attr-spec*.

NOTE 4.51

The name of the parent type might be a local name introduced via renaming in a USE statement.

3 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.

4 An abstract type is a type that has the ABSTRACT attribute.

NOTE 4.52

The DEFERRED attribute (4.5.5) 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:

```
TYPE, ABSTRACT :: FILE_HANDLE
CONTAINS
```

NOTE 4.52 (cont.)

```
    PROCEDURE(OPEN_FILE), DEFERRED, PASS(HANDLE) :: OPEN
    ...
  END TYPE
```

For a more elaborate example see C.1.4.

4.5.7.2 Inheritance

1 An extended type includes all of the type parameters, all of the components, and the nonoverridden (4.5.7.3) 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 4.53

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 4.54

A 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.

2 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. It has the accessibility of the parent type. Components of the parent component are inheritance associated (16.5.4) 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.

NOTE 4.55

A component or type parameter declared in an extended type shall not have the same name as any accessible component or type parameter of its parent type.

NOTE 4.56

Examples:

```
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
```

4.5.7.3 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 (12.3.3).
- If the overridden type-bound procedure is PUBLIC then the overriding one shall not be PRIVATE.

NOTE 4.57

The following is an example of procedure overriding, expanding on the example in Note 4.43.

```
TYPE, EXTENDS (POINT) :: POINT_3D
  REAL :: Z
CONTAINS
  PROCEDURE, PASS :: LENGTH => POINT_3D_LENGTH
END TYPE POINT_3D
...
```

and in the *module-subprogram-part* of the same module:

```
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.'
  STOP
END FUNCTION POINT_3D_LENGTH
```

3 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 12.4.3.5.5.

4 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.

4.5.8 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.

4.5.9 Derived-type specifier

1 A derived-type specifier is used in several contexts to specify a particular derived type and type parameters.

R454 *derived-type-spec* **is** *type-name* [(*type-param-spec-list*)]

R455 *type-param-spec* **is** [*keyword* =] *type-param-value*

C488 (R454) *type-name* shall be the name of an accessible derived type.

C489 (R454) *type-param-spec-list* shall appear only if the type is parameterized.

C490 (R454) 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.

C491 (R455) 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*.

C492 (R455) Each *keyword* shall be the name of a parameter of the type.

C493 (R455) 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 (4.5.3.2). 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 (7.2.1.3) 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.

4.5.10 Construction of derived-type values

1 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.

R456 *structure-constructor* **is** *derived-type-spec* ([*component-spec-list*])

R457 *component-spec* **is** [*keyword* =] *component-data-source*

R458 *component-data-source* **is** *expr*
or *data-target*
or *proc-target*

C494 (R456) The *derived-type-spec* shall not specify an abstract type (4.5.7).

C495 (R456) At most one *component-spec* shall be provided for a component.

C496 (R456) 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.

C497 (R456) 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.

C498 (R457) The *keyword*= may be omitted from a *component-spec* only if the *keyword*= has been omitted from each preceding *component-spec* in the constructor.

C499 (R457) Each *keyword* shall be the name of a component of the type.

C4100 (R456) 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.

C4101 (R456) 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 (12.5.5.2).

C4102 (R458) A *data-target* shall correspond to a data pointer component; a *proc-target* shall correspond to a procedure pointer component.

C4103 (R458) A *data-target* shall have the same rank as its corresponding component.

NOTE 4.58

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*.

2 In the absence of a component keyword, each *component-data-source* is assigned to the corresponding component in component order (4.5.4.7). 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 (7.2.1.2), as specified in Table 7.8. If necessary, each value of intrinsic type is converted according to the rules of intrinsic assignment (7.2.1.3) 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.

3 If a component with default initialization has no corresponding *component-data-source*, then the default initialization is applied to that component. If an allocatable component has no corresponding *component-data-source*, then that component has an allocation status of unallocated.

NOTE 4.59

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 4.56:

```
! 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.
```

4 A structure constructor shall not appear before the referenced type is defined.

NOTE 4.60

This example illustrates a derived-type constant expression using a derived type defined in Note 4.16:

```
PERSON (21, 'JOHN SMITH')
```

This could also be written as

```
PERSON (NAME = 'JOHN SMITH', AGE = 21)
```

NOTE 4.61

An example constructor using the derived type GENERAL_POINT defined in Note 4.23 is

NOTE 4.61 (cont.)

```
general_point(dim=3) ( [ 1., 2., 3. ] )
```

5 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 (7.2.2). 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 4.62

For example, if the variable TEXT were declared (5.2) to be

```
CHARACTER, DIMENSION (1:400), TARGET :: TEXT
```

and BIBLIO were declared using the derived-type definition REFERENCE in Note 4.31

```
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.

6 If a component of a derived type is allocatable, the corresponding constructor expression shall either 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 expression is an allocatable entity, the corresponding component of the constructor has the same allocation status as that allocatable entity and, if it is allocated, the same dynamic type, bounds, and value; if a length parameter of the component is deferred, its value is the same as the corresponding parameter of the expression. Otherwise the corresponding component of the constructor has an allocation status of allocated and has the same bounds and value as the expression.

NOTE 4.63

When the constructor is an actual argument, the allocation status of the allocatable component is available through the associated dummy argument.

4.5.11 Derived-type operations and assignment

1 Intrinsic assignment of derived-type entities is described in 7.2.1. This part of ISO/IEC 1539 does not specify any intrinsic operations on derived-type entities. Any operation on derived-type entities or defined assignment (7.2.1.4) for derived-type entities shall be defined explicitly by a function or a subroutine, and a generic interface (4.5.5, 12.4.3.2).

4.6 Enumerations and enumerators

1 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.

R459 *enum-def* **is** *enum-def-stmt*
enumerator-def-stmt

[*enumerator-def-stmt*] ...
end-enum-stmt

R460	*enum-def-stmt*	**is**	ENUM, BIND(C)
R461	*enumerator-def-stmt*	**is**	ENUMERATOR [::] *enumerator-list*
R462	*enumerator*	**is**	*named-constant* [= *scalar-int-constant-expr*]
R463	*end-enum-stmt*	**is**	END ENUM

C4104 (R461) If = appears in an *enumerator*, a double-colon separator shall appear before the *enumerator-list*.

2 For an enumeration, the kind is selected such that an integer type with that kind is interoperable (15.3.2) 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*.

3 The companion processor (2.5.7) 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 4.64

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 enumerators cannot be represented in this signed integer type. The types of any such enumerators will be interoperable with the type declared in the C enumeration.

NOTE 4.65

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 4.66

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 part of ISO/IEC 1539 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.

4 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 (7.2) 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 in the *enum-def*.

NOTE 4.67

Example of an enumeration definition:

```
ENUM, BIND(C)
```

NOTE 4.67 (cont.)

```
  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
```

NOTE 4.68

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.

4.7 Binary, octal, and hexadecimal literal constants

1 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.

R464 *boz-literal-constant* **is** *binary-constant*
or *octal-constant*
or *hex-constant*

R465 *binary-constant* **is** B ' *digit* [*digit*] ... '
or B " *digit* [*digit*] ... "

C4105 (R465) *digit* shall have one of the values 0 or 1.

R466 *octal-constant* **is** O ' *digit* [*digit*] ... '
or O " *digit* [*digit*] ... "

C4106 (R466) *digit* shall have one of the values 0 through 7.

R467 *hex-constant* **is** Z ' *hex-digit* [*hex-digit*] ... '
or Z " *hex-digit* [*hex-digit*] ... "

R468 *hex-digit* **is** *digit*
or A
or B
or C
or D
or E
or F

2 The *hex-digit*s 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 13.3, 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 (13.7.165) with an argument that is a real or integer scalar of any kind supported by the processor.

C4107 (R464) A *boz-literal-constant* shall appear only as a *data-stmt-constant* in a DATA statement, or where explicitly allowed in subclause 13.7 as an actual argument of an intrinsic procedure.

4.8 Construction of array values

1 An array constructor constructs a rank-one array value from a sequence of scalar values, array values, and implied DO loops.

R469 *array-constructor* **is** (/ *ac-spec* /)
or *lbracket ac-spec rbracket*

R470 *ac-spec* **is** *type-spec* ::
or [*type-spec* ::] *ac-value-list*

R471 *lbracket* **is** [

R472 *rbracket* **is**]

R473 *ac-value* **is** *expr*
or *ac-implied-do*

R474 *ac-implied-do* **is** (*ac-value-list* , *ac-implied-do-control*)

R475 *ac-implied-do-control* **is** [*integer-type-spec* ::] *ac-do-variable* = *scalar-int-expr* , ■
■ *scalar-int-expr* [, *scalar-int-expr*]

R476 *ac-do-variable* **is** *do-variable*

C4108 (R470) If *type-spec* is omitted, each *ac-value* expression in the *array-constructor* shall have the same declared type and kind type parameters.

C4109 (R470) 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 7.8.

C4110 (R470) 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*.

C4111 (R473) An *ac-value* shall not be unlimited polymorphic.

C4112 (R473) The declared type of an *ac-value* shall not be abstract.

C4113 (R474) 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.

3 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 (7.2.1.3).

4 The dynamic type of an array constructor is the same as its declared type.

5 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.

6 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 (6.5.3.2), 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 (8.1.6.4).

7 For an *ac-implied-do*, the loop initialization and execution is the same as for a DO construct.

8 An empty sequence forms a zero-sized array.

NOTE 4.69

A one-dimensional array can be reshaped into any allowable array shape using the intrinsic function RESHAPE (13.7.145). 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 3×2 array of values:

```
2.0   3.2
4.5   4.01
4.5   6.5
```

NOTE 4.70

Examples of array constructors containing an implied DO are:

```
(/ (I, I = 1, 1075) /)
```

and

```
[ 3.6, (3.6 / I, I = 1, N) ]
```

NOTE 4.71

Using the type definition for PERSON in Note 4.16, an example of the construction of a derived-type array value is:

```
[ PERSON (40, 'SMITH'), PERSON (20, 'JONES') ]
```

NOTE 4.72

Using the type definition for LINE in Note 4.27, 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 4.73

Examples of zero-size array constructors are:

```
   [ INTEGER :: ]
```

NOTE 4.73 (cont.)

```
[ ( I, I = 1, 0) ]
```

NOTE 4.74

An example of an array constructor that specifies a length type parameter:

```
[ 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.

(Blank page)

5 Attribute declarations and specifications

5.1 Attributes of procedures and data objects

1 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 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 (5.7). All of its attributes may be specified in a type declaration statement or individually in separate specification statements.

2 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.

3 A subroutine does not have a type, rank, or type parameters, but may have other attributes that determine the uses of the subroutine.

5.2 Type declaration statement

R501 *type-declaration-stmt* **is** *declaration-type-spec* [[, *attr-spec*] ... ::] *entity-decl-list*

1 The type declaration statement specifies the 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*.

R502 *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

C501 (R501) The same *attr-spec* shall not appear more than once in a given *type-declaration-stmt*.

C502 (R501) If a *language-binding-spec* with a NAME= specifier appears, the *entity-decl-list* shall consist of a single *entity-decl*.

C503 (R501) If a *language-binding-spec* is specified, the *entity-decl-list* shall not contain any procedure names.

2 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*.

R503 *entity-decl* **is** *object-name* [(*array-spec*)] ■
■ [*lbracket coarray-spec rbracket*] ■
■ [* *char-length*] [*initialization*]
or *function-name* [* *char-length*]

C504 (R503) If the entity is not of type character, * *char-length* shall not appear.

C505 (R501) If *initialization* appears, a double-colon separator shall appear before the *entity-decl-list*.

C506 (R501) If the PARAMETER keyword appears, *initialization* shall appear in each *entity-decl*.

C507 (R503) 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 object.

C508 (R503) The *function-name* shall be the name of an external function, an intrinsic function, a dummy function, a procedure pointer, or a statement function.

R504 *object-name* **is** *name*

C509 (R504) The *object-name* shall be the name of a data object.

R505 *initialization* **is** = *constant-expr*
or => *null-init*
or => *initial-data-target*

R506 *null-init* **is** *function-reference*

C510 (R503) If => appears in *initialization*, the entity shall have the POINTER attribute. If = appears in *initialization*, the entity shall not have the POINTER attribute.

C511 (R503) If *initial-data-target* appears, *object-name* shall be data-pointer-initialization compatible with it (4.5.4.6).

C512 (R506) 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 13.6. 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 (7.2.1.2);
- 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 5.1

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 5.1 (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 4.23.)

5.3 Automatic data objects

1 An automatic data object is a 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.

C513 An automatic 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.

5.4 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 (7.2.1.3) 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.

5.5 Attributes

5.5.1 Attribute specification

1 An attribute may be explicitly specified by an *attr-spec* in a type declaration statement or by an attribute specification statement (5.6). The following constraints apply to attributes.

C514 An entity shall not be explicitly given any attribute more than once in a scoping unit.

C515 An *array-spec* for a nonallocatable nonpointer function result shall be an *explicit-shape-spec-list*.

C516 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*.

5.5.2 Accessibility attribute

1 The accessibility attribute specifies the accessibility of an entity via a particular identifier.

R507 *access-spec* **is** PUBLIC
or PRIVATE

C517 An *access-spec* shall appear only in the *specification-part* of a module.

2 Identifiers that are specified in a module or accessible in that module by use association have either the PUBLIC attribute or PRIVATE attribute. Identifiers for which an *access-spec* is not explicitly specified in that module have the default accessibility attribute for that module. The default accessibility attribute for a module is PUBLIC unless it has been changed by a PRIVATE statement (5.6.1). Only identifiers that have the PUBLIC attribute in that module are available to be accessed from that module by use association.

NOTE 5.2

In order for an identifier to be accessed by use association, it must have 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 for use association from that module.

NOTE 5.3

An example of an accessibility specification is:

```
REAL, PRIVATE :: X, Y, Z
```

5.5.3 ALLOCATABLE attribute

1 An entity with the ALLOCATABLE attribute is a variable for which space is allocated by an ALLOCATE statement (6.7.1) or by an intrinsic assignment statement (7.2.1.2).

5.5.4 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 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 (9.6.2.5) or a pending communication affector (15.10.4).

3 Use of a variable in an asynchronous data transfer statement can imply the ASYNCHRONOUS attribute; see subclause 9.6.2.5.

4 An object with the ASYNCHRONOUS attribute may be associated with an object that does not have the ASYNCHRONOUS attribute, including by use (11.2.2) or host association (16.5.1.4). 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 5.4

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.

5.5.5 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 (15.9).

R508 *language-binding-spec* **is** BIND (C [, NAME = *scalar-default-char-constant-expr*])

C518 An entity with the BIND attribute shall be a common block, variable, type, or procedure.

C519 A variable with the BIND attribute shall be declared in the specification part of a module.

C520 A variable with the BIND attribute shall be interoperable (15.3).

C521 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 5.5

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 variable or common block implies the SAVE attribute, which may be confirmed by explicit specification.

5.5.6 CODIMENSION attribute

5.5.6.1 General

1 The CODIMENSION attribute specifies that an entity is a coarray. The *coarray-spec* specifies its corank or corank and cobounds.

R509 *coarray-spec* **is** *deferred-coshape-spec-list*
or *explicit-coshape-spec*

C522 The sum of the rank and corank of an entity shall not exceed fifteen.

C523 A coarray shall be a component or a variable that is not a function result.

C524 A coarray shall not be of type C_PTR or C_FUNPTR (15.3.3).

C525 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.

C526 A coarray or an object with a coarray ultimate component shall be a dummy argument or have the ALLOCATABLE or SAVE attribute.

C527 A coarray shall not be a dummy argument of a procedure that has a *proc-language-binding-spec*.

NOTE 5.6

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 5.7

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 5.8

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
```

5.5.6.2 Allocatable coarray

1 A coarray with the ALLOCATABLE attribute has a specified corank, but its cobounds are determined by allocation or argument association.

R510 *deferred-coshape-spec* **is** :

C528 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 13.1.

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.

5.5.6.3 Explicit-coshape coarray

1 An explicit-coshape coarray is a named coarray that has its corank and cobounds declared by an *explicit-coshape-spec*.

R511 *explicit-coshape-spec* **is** [[*lower-cobound* :] *upper-cobound*,]... ■
■ [*lower-cobound* :] *

C529 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-cobound*s.

R512 *lower-cobound* **is** *specification-expr*

R513 *upper-cobound* **is** *specification-expr*

C530 (R511) A *lower-cobound* or *upper-cobound* that is not a constant expression shall appear only in a subprogram, 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.

5.5.7 CONTIGUOUS attribute

C531 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 (6.5.3) 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* (6.4.1),
    (f) only its final *part-ref* has nonzero rank, and
    (g) it is not the real or imaginary part (6.4.4) 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.

4 It is processor dependent whether any other object is contiguous.

NOTE 5.9

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 5.10

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:

```
   REAL, POINTER, CONTIGUOUS         :: SPTR(:)
   REAL, CONTIGUOUS, DIMENSION(:,:) :: D
```

5.5.8 DIMENSION attribute

5.5.8.1 General

1 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*.

R514 *dimension-spec* **is** DIMENSION (*array-spec*)

R515 *array-spec* **is** *explicit-shape-spec-list*
or *assumed-shape-spec-list*
or *deferred-shape-spec-list*
or *assumed-size-spec*
or *implied-shape-spec*
or *implied-shape-or-assumed-size-spec*
or *assumed-rank-spec*

NOTE 5.11

The maximum rank of an entity is fifteen minus the corank.

NOTE 5.12

Examples of DIMENSION attribute specifications are:

```
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
```

5.5.8.2 Explicit-shape array

R516 *explicit-shape-spec* **is** [*lower-bound* :] *upper-bound*

R517 *lower-bound* **is** *specification-expr*

R518 *upper-bound* **is** *specification-expr*

C532 (R516) An *explicit-shape-spec* whose bounds are not constant expressions shall appear only in a subprogram, derived type definition, BLOCK construct, or interface body.

1 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-spec*s.

2 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.

3 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.

5.5.8.3 Assumed-shape array

1 An assumed-shape array is a nonallocatable nonpointer dummy argument array that takes its shape from its effective argument.

R519 *assumed-shape-spec* **is** [*lower-bound*] :

2 The rank is equal to the number of colons in the *assumed-shape-spec-list*.

3 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.

5.5.8.4 Deferred-shape array

1 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.)

R520 *deferred-shape-spec* **is** :

C533 An array with the POINTER or ALLOCATABLE attribute shall have an *array-spec* that is a *deferred-shape-spec-list*.

2 The rank is equal to the number of colons in the *deferred-shape-spec-list*.

3 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 13.1.

4 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.

5 The bounds of each dimension of an associated array pointer, and hence its shape, may be specified

- in an ALLOCATE statement (6.7.1) when the target is allocated,
- by pointer assignment (7.2.2), or
- if it is a dummy argument, by argument association with a nonpointer actual argument or an associated pointer effective argument.

6 The bounds of an array pointer or allocatable array are unaffected by any subsequent redefinition or undefinition of variables on which the bounds' expressions depend.

5.5.8.5 Assumed-size array

1 An assumed-size array is a dummy argument array whose size is assumed from that of its effective argument. 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*.

R521 *assumed-implied-spec* **is** [*lower-bound* :] *

R522 *assumed-size-spec* **is** *explicit-shape-spec-list*, *assumed-implied-spec*

C534 An object whose array bounds are specified by an *assumed-size-spec* shall be a dummy data object.

C535 An assumed-size array with the INTENT (OUT) attribute shall not be polymorphic, finalizable, of a type with an allocatable ultimate component, or of a type for which default initialization is specified.

R523 *implied-shape-or-assumed-size-spec* **is** assumed-implied-spec

C536 An object whose array bounds are specified by an *implied-shape-or-assumed-size-spec* shall be a dummy data object or a named constant.

2 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 (6.5.3.2) 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 (6.4.1), and if it begins at character storage unit t of an array with c character storage units, the size of the dummy array is MAX (INT $((c - t + 1)/e)$, 0), 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 MAX (INT (l/e), 0), where e is the length of an element in the dummy character array and l is the length of the actual argument.

3 The rank is equal to one plus the number of *explicit-shape-spec*s.

4 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.

5 If a list of *explicit-shape-spec*s 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 (6.5.3.3). The upper bound shall not be omitted from a subscript triplet in the last dimension.

6 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 undefinition of any variable during execution of the procedure.

5.5.8.6 Implied-shape array

1 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-size-spec* or an *implied-shape-spec*.

R524 *implied-shape-spec* **is** *assumed-implied-spec*, *assumed-implied-spec-list*

C537 An implied-shape array shall be a named constant.

2 The rank of an implied-shape array is the number of *assumed-implied-spec*s in its *array-spec*.

3 The extent of each dimension of an implied-shape array is the same as the extent of the corresponding dimension of the *constant-expr*. The lower bound of each dimension is *lower-bound*, if it appears, and 1 otherwise; the upper bound is one less than the sum of the lower bound and the extent.

5.5.8.7 Assumed-rank entity

1 An assumed-rank entity is a dummy data object whose rank is assumed from its effective argument; this rank can be zero. An assumed-rank entity is declared with an *array-spec* that is an *assumed-rank-spec*.

R525 *assumed-rank-spec* **is** ..

C538 An assumed-rank entity shall be a dummy data object that does not have the CODIMENSION or VALUE attribute.

C539 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 from the intrinsic module ISO_C_BINDING (15.2.3.6), or the first dummy argument of an intrinsic inquiry function.

C540 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.

5.5.9 EXTERNAL attribute

1 The EXTERNAL attribute specifies that an entity is an external procedure, dummy procedure, procedure pointer, or block data subprogram.

C541 An entity shall not have both the EXTERNAL attribute and the INTRINSIC attribute.

C542 In an external subprogram, the EXTERNAL attribute shall not be specified for a procedure defined by the subprogram.

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 5.13

> The EXTERNAL attribute can be specified in a type declaration statement, by an interface body (12.4.3.2), by an EXTERNAL statement (12.4.3.6), or by a procedure declaration statement (12.4.3.7).

5.5.10 INTENT attribute

1 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.

R526 *intent-spec* **is** IN
or OUT

or INOUT

C543 An entity with the INTENT attribute shall be a dummy data object or a dummy procedure pointer.

C544 (R526) A nonpointer object with the INTENT (IN) attribute shall not appear in a variable definition context (16.6.7).

C545 A pointer with the INTENT (IN) attribute shall not appear in a pointer association context (16.6.8).

C546 An INTENT (OUT) dummy argument of a nonintrinsic procedure shall not be an allocatable coarray or have a subobject that is an allocatable coarray.

C547 An entity with the INTENT (OUT) attribute shall not be of type LOCK_TYPE (13.8.2.16) of the intrinsic module ISO_FORTRAN_ENV or have a subcomponent of this type.

2 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 (16.5.2.5).

3 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 (4.5.4.6). 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. Any undefinition 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.

4 The INTENT (INOUT) attribute for a nonpointer dummy argument specifies that any actual argument that corresponds to the dummy argument shall be definable. The INTENT (INOUT) attribute for a pointer dummy argument specifies that any actual argument that corresponds to the dummy argument shall be a pointer variable.

NOTE 5.14

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 (6.7.3.2). 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.

5 If no INTENT attribute is specified for a dummy argument, its use is subject to the limitations of its effective argument (12.5.2).

NOTE 5.15

An example of INTENT specification is:

```
SUBROUTINE MOVE (FROM, TO)
   USE PERSON_MODULE
   TYPE (PERSON), INTENT (IN) :: FROM
   TYPE (PERSON), INTENT (OUT) :: TO
```

6 If an object has an INTENT attribute, then all of its subobjects have the same INTENT attribute.

NOTE 5.16

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

NOTE 5.16 (cont.)

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 5.17

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.

5.5.11 INTRINSIC attribute

1 The INTRINSIC attribute specifies that the entity is an intrinsic procedure. The procedure name may be a generic name (13.5), a specific name (13.6), or both.

2 If the specific name of an intrinsic procedure (13.6) 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 13.3 is not permitted to be used as an actual argument (C1240).

C548 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 (12.4.3.2) accessible in the same scoping unit, the procedures in the interfaces and the generic intrinsic procedure shall all be functions or all be subroutines.

5.5.12 OPTIONAL attribute

1 The OPTIONAL attribute specifies that the dummy argument need not have a corresponding actual argument in a reference to the procedure (12.5.2.12).

C549 An entity with the OPTIONAL attribute shall be a dummy argument.

NOTE 5.18

The intrinsic function PRESENT (13.7.134) can be used to determine whether an optional dummy argument has a corresponding actual argument.

5.5.13 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.

C550 An entity with the PARAMETER attribute shall not be a variable, a coarray, or a procedure.

C551 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 in the same statement, defined in a prior statement, or made accessible by use or host association.

NOTE 5.19

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 ( ))
```

5.5.14 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. Procedure pointers are described in 12.4.3.7.

C552 An entity with the POINTER attribute shall not have the ALLOCATABLE, INTRINSIC, or TARGET attribute, and shall not be a coarray.

C553 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 5.20

Examples of POINTER attribute specifications are:

```
TYPE (NODE), POINTER :: CURRENT, TAIL
REAL, DIMENSION (:, :), POINTER :: IN, OUT, SWAP
```

For a more elaborate example see C.2.1.

5.5.15 PROTECTED attribute

1 The PROTECTED attribute imposes limitations on the usage of module entities.

C554 The PROTECTED attribute shall be specified only in the specification part of a module.

C555 An entity with the PROTECTED attribute shall be a procedure pointer or variable.

C556 An entity with the PROTECTED attribute shall not be in a common block.

C557 A nonpointer object that has the PROTECTED attribute and is accessed by use association shall not appear in a variable definition context (16.6.7) or as the *data-target* or *proc-target* in a *pointer-assignment-stmt*.

C558 A pointer that has the PROTECTED attribute and is accessed by use association shall not appear in a pointer association context (16.6.8).

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 (16.5.2.5) or if its target becomes undefined 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 5.21

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.

5.5.16 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 (16.5.2.5(6)). If it is a local variable of a subprogram it is shared by all instances (12.6.2.4) 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 (16.5.2.5(7)). If the BLOCK construct is within a subprogram the variable is shared by all instances (12.6.2.4) of the subprogram.

3 Giving a common block the SAVE attribute confers the attribute on all entities in the common block.

C559 An entity with the SAVE attribute shall be a common block, variable, or procedure pointer.

C560 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.

4 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.

5.5.17 TARGET attribute

1 The TARGET attribute specifies that a data object may have a pointer associated with it (7.2.2). An object without the TARGET attribute shall not have a pointer associated with it.

C561 An entity with the TARGET attribute shall be a variable.

C562 An entity with the TARGET attribute shall not have the POINTER attribute.

NOTE 5.22

In addition to variables explicitly declared to have the TARGET attribute, the objects created by allocation of pointers (6.7.1.4) have the TARGET attribute.

2 If an object has the TARGET attribute, then all of its nonpointer subobjects also have the TARGET attribute.

NOTE 5.23

Examples of TARGET attribute specifications are:

```
TYPE (NODE), TARGET :: HEAD
REAL, DIMENSION (1000, 1000), TARGET :: A, B
```

For a more elaborate example see C.2.2.

NOTE 5.24

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.

5.5.18 VALUE attribute

1 The VALUE attribute specifies a type of argument association (12.5.2.4) for a dummy argument.

C563 An entity with the VALUE attribute shall be a dummy data object that is not an assumed-size array or a coarray, and does not have a coarray ultimate component.

C564 An entity with the VALUE attribute shall not have the ALLOCATABLE, INTENT (INOUT), INTENT (OUT), POINTER, or VOLATILE attributes.

5.5.19 VOLATILE attribute

1 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.

C565 An entity with the VOLATILE attribute shall be a variable that is not an INTENT (IN) dummy argument.

C566 The VOLATILE attribute shall not be specified for a coarray that is accessed by use (11.2.2) or host (16.5.1.4) association.

C567 Within a BLOCK construct (8.1.4), the VOLATILE attribute shall not be specified for a coarray that is not a construct entity (16.4) of that construct.

2 A noncoarray object that has the VOLATILE attribute may be associated with an object that does not have the VOLATILE attribute, including by use (11.2.2) or host association (16.5.1.4). 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 12.5.2.8. The relationship between between coarrays, the VOLATILE attribute, and pointer association is described in 7.2.2.3.

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.

4 If an object has the VOLATILE attribute, then all of its subobjects also have the VOLATILE attribute.

5 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.

5.6 Attribute specification statements

5.6.1 Accessibility statement

R527	*access-stmt*	**is**	*access-spec* [[::] *access-id-list*]
R528	*access-id*	**is**	*access-name*
		or	*generic-spec*

C568 (R527) An *access-stmt* shall appear only in the *specification-part* of a module. Only one accessibility statement with an omitted *access-id-list* is permitted in the *specification-part* of a module.

C569 (R528) Each *access-name* shall be the name of a module, variable, procedure, derived type, named constant, or namelist group.

C570 A module whose name appears in an *access-stmt* shall be referenced by a USE statement in the scoping unit that contains the *access-stmt*.

C571 The name of a module shall appear at most once in all of the *access-stmt*s in a module.

1 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. Appearance of a module name in an *access-stmt* specifies the default accessibility of the identifiers of entities accessed from that module. 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
```

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.

NOTE 5.25

Examples of accessibility statements are:

```
MODULE EX
   PRIVATE
   PUBLIC :: A, B, C, ASSIGNMENT (=), OPERATOR (+)
```

NOTE 5.26

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
```

5.6.2 ALLOCATABLE statement

R529 *allocatable-stmt* **is** ALLOCATABLE [::] *allocatable-decl-list*

R530 *allocatable-decl* **is** *object-name* [(*array-spec*)] ■
■ [*lbracket coarray-spec rbracket*]

1 The ALLOCATABLE statement specifies the ALLOCATABLE attribute (5.5.3) for a list of objects.

NOTE 5.27

An example of an ALLOCATABLE statement is:

```
REAL A, B (:), SCALAR
ALLOCATABLE :: A (:, :), B, SCALAR
```

5.6.3 ASYNCHRONOUS statement

R531 *asynchronous-stmt* **is** ASYNCHRONOUS [::] *object-name-list*

1 The ASYNCHRONOUS statement specifies the ASYNCHRONOUS attribute (5.5.4) for a list of objects.

5.6.4 BIND statement

R532 *bind-stmt* **is** *language-binding-spec* [::] *bind-entity-list*

R533 *bind-entity* **is** *entity-name*
or / *common-block-name* /

C572 (R532) 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.

5.6.5 CODIMENSION statement

R534 *codimension-stmt* **is** CODIMENSION [::] *codimension-decl-list*

R535 *codimension-decl* **is** *coarray-name lbracket coarray-spec rbracket*

1 The CODIMENSION statement specifies the CODIMENSION attribute (5.5.6) for a list of objects.

NOTE 5.28

An example of a CODIMENSION statement is:

```
CODIMENSION a[*], b[3,*], c[:]
```

5.6.6 CONTIGUOUS statement

R536 *contiguous-stmt* **is** CONTIGUOUS [::] *object-name-list*

1 The CONTIGUOUS statement specifies the CONTIGUOUS attribute (5.5.7) for a list of objects.

5.6.7 DATA statement

R537 *data-stmt* **is** DATA *data-stmt-set* [[,] *data-stmt-set*] ...

1 The DATA statement specifies explicit initialization (5.4).

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.

R538 *data-stmt-set* **is** *data-stmt-object-list* / *data-stmt-value-list* /

R539 *data-stmt-object* **is** *variable*
or *data-implied-do*

R540 *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*])

R541 *data-i-do-object* **is** *array-element*
or *scalar-structure-component*
or *data-implied-do*

R542 *data-i-do-variable* **is** *do-variable*

C573 A *data-stmt-object* or *data-i-do-object* shall not be a coindexed variable.

C574 (R539) 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.

C575 (R539) 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 object, or an allocatable variable.

C576 (R539) 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*.

C577 (R541) The *array-element* shall be a variable.

C578 (R541) The *scalar-structure-component* shall be a variable.

C579 (R541) The *scalar-structure-component* shall contain at least one *part-ref* that contains a *subscript-list*.

C580 (R541) 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*.

R543 *data-stmt-value* **is** [*data-stmt-repeat* *] *data-stmt-constant*

R544 *data-stmt-repeat* **is** *scalar-int-constant*
or *scalar-int-constant-subobject*

C581 (R544) The *data-stmt-repeat* shall be positive or zero. If the *data-stmt-repeat* is a named constant, it shall have been declared previously in the scoping unit or made accessible by use or host association.

R545 *data-stmt-constant* **is** *scalar-constant*
or *scalar-constant-subobject*
or *signed-int-literal-constant*
or *signed-real-literal-constant*
or *null-init*
or *initial-data-target*
or *structure-constructor*

C582 (R545) If a DATA statement constant value is a named constant or a structure constructor, the named constant or derived type shall have been declared previously in the scoping unit or accessed by use or host association.

C583 (R545) If a *data-stmt-constant* is a *structure-constructor*, it shall be a constant expression.

R546 *int-constant-subobject* **is** *constant-subobject*

C584 (R546) *int-constant-subobject* shall be of type integer.

R547 *constant-subobject* **is** *designator*

C585 (R547) *constant-subobject* shall be a subobject of a constant.

C586 (R547) Any subscript, substring starting point, or substring ending point shall be a constant expression.

5 The *data-stmt-object-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 (6.5.3.2). 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 (8.1.6.4).

6 The *data-stmt-value-list* is expanded to form a sequence of *data-stmt-constant*s. A *data-stmt-repeat* indicates the number of times the following *data-stmt-constant* is to be included in the sequence; omission of a *data-stmt-repeat* has the effect of a repeat factor of 1.

7 A zero-sized array or a *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 *data-stmt-constant* with a repeat factor of zero contributes no *data-stmt-constant*s to the expanded sequence of scalar *data-stmt-constant*s.

8 The expanded sequences of variables and *data-stmt-constant*s are in one-to-one correspondence. Each *data-stmt-constant* specifies the initial value, initial data target, or *null-init* for the corresponding variable. The lengths of

the two expanded sequences shall be the same.

9 A *data-stmt-constant* shall be *null-init* or *initial-data-target* if and only if the corresponding *data-stmt-object* has the POINTER attribute. If *data-stmt-constant* is *null-init*, the initial association status of the corresponding data statement object is disassociated. If *data-stmt-constant* is *initial-data-target* the corresponding data statement object shall be data-pointer-initialization compatible (4.5.4.6) with the initial data target; the data statement object is initially associated with the target.

10 A *data-stmt-constant* other than *boz-literal-constant*, *null-init*, or *initial-data-target* shall be compatible with its corresponding variable according to the rules of intrinsic assignment (7.2.1.2). The variable is initially defined with the value specified by the *data-stmt-constant*; if necessary, the value is converted according to the rules of intrinsic assignment (7.2.1.3) to a value that agrees in type, type parameters, and shape with the variable.

11 If a *data-stmt-constant* is a *boz-literal-constant*, the corresponding variable shall be of type integer. The *boz-literal-constant* is treated as if it were converted by the intrinsic function INT (13.7.82) to type integer with the kind type parameter of the variable.

NOTE 5.29

Examples of DATA statements are:

```
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' /
```

The character variable NAME is initialized with the value 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 MILES are initialized to zero. The two-dimensional array SKEW is initialized so that the lower triangle of SKEW is zero and the strict upper triangle is one. The structures MYNAME and YOURNAME are declared using the derived type PERSON from Note 4.16. The pointer HEAD_OF_LIST is declared using the derived type NODE from Note 4.36; it is initially disassociated. MYNAME is initialized by a structure constructor. YOURNAME is initialized by supplying a separate value for each component.

5.6.8 DIMENSION statement

R548 *dimension-stmt* **is** DIMENSION [::] *array-name* (*array-spec*) ■
■ [, *array-name* (*array-spec*)] ...

1 The DIMENSION statement specifies the DIMENSION attribute (5.5.8) for a list of objects.

NOTE 5.30

An example of a DIMENSION statement is:

```
DIMENSION A (10), B (10, 70), C (:)
```

5.6.9 INTENT statement

R549 *intent-stmt* **is** INTENT (*intent-spec*) [::] *dummy-arg-name-list*

1 The INTENT statement specifies the INTENT attribute (5.5.10) for the dummy arguments in the list.

NOTE 5.31

An example of an INTENT statement is:

```
SUBROUTINE EX (A, B)
   INTENT (INOUT) :: A, B
```

5.6.10 OPTIONAL statement

R550 *optional-stmt* **is** OPTIONAL [::] *dummy-arg-name-list*

1 The OPTIONAL statement specifies the OPTIONAL attribute (5.5.12) for the dummy arguments in the list.

NOTE 5.32

An example of an OPTIONAL statement is:

```
SUBROUTINE EX (A, B)
   OPTIONAL :: B
```

5.6.11 PARAMETER statement

1 The PARAMETER statement specifies the PARAMETER attribute (5.5.13) and the values for the named constants in the list.

R551 *parameter-stmt* **is** PARAMETER (*named-constant-def-list*)

R552 *named-constant-def* **is** *named-constant* = *constant-expr*

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 (5.7). 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 (7.2.1.2). 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 (7.2.1.3) to a value that agrees in type, type parameters, and shape with the named constant.

NOTE 5.33

An example of a PARAMETER statement is:

```
PARAMETER (MODULUS = MOD (28, 3), NUMBER_OF_SENATORS = 100)
```

5.6.12 POINTER statement

R553 *pointer-stmt* **is** POINTER [::] *pointer-decl-list*

R554 *pointer-decl* **is** *object-name* [(*deferred-shape-spec-list*)]
or *proc-entity-name*

C587 A *proc-entity-name* shall have the EXTERNAL attribute.

1 The POINTER statement specifies the POINTER attribute (5.5.14) for a list of entities.

NOTE 5.34

An example of a POINTER statement is:

```
TYPE (NODE) :: CURRENT
POINTER :: CURRENT, A (:, :)
```

5.6.13 PROTECTED statement

R555 *protected-stmt* **is** PROTECTED [::] *entity-name-list*

1 The PROTECTED statement specifies the PROTECTED attribute (5.5.15) for a list of entities.

5.6.14 SAVE statement

R556 *save-stmt* **is** SAVE [[::] *saved-entity-list*]

R557 *saved-entity* **is** *object-name*
or *proc-pointer-name*
or / *common-block-name* /

R558 *proc-pointer-name* **is** *name*

C588 (R556) 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.

C589 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 (5.5.16) 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 5.35

An example of a SAVE statement is:

```
SAVE A, B, C, / BLOCKA /, D
```

5.6.15 TARGET statement

R559 *target-stmt* **is** TARGET [::] *target-decl-list*
R560 *target-decl* **is** *object-name* [(*array-spec*)] ■
■ [*lbracket coarray-spec rbracket*]

1 The TARGET statement specifies the TARGET attribute (5.5.17) for a list of objects.

NOTE 5.36

An example of a TARGET statement is:

```
TARGET :: A (1000, 1000), B
```

5.6.16 VALUE statement

R561 *value-stmt* **is** VALUE [::] *dummy-arg-name-list*

1 The VALUE statement specifies the VALUE attribute (5.5.18) for a list of dummy arguments.

5.6.17 VOLATILE statement

R562 *volatile-stmt* **is** VOLATILE [::] *object-name-list*

1 The VOLATILE statement specifies the VOLATILE attribute (5.5.19) for a list of objects.

5.7 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. Alternatively, it may indicate that no implicit typing rules are to apply in a particular scoping unit.

R563 *implicit-stmt* **is** IMPLICIT *implicit-spec-list*
or IMPLICIT NONE [([*implicit-none-spec-list*])]

R564 *implicit-spec* **is** *declaration-type-spec* (*letter-spec-list*)

R565 *letter-spec* **is** *letter* [– *letter*]

R566 *implicit-none-spec* **is** EXTERNAL
or TYPE

C590 (R563) If IMPLICIT NONE is specified 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.

C591 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.

C592 (R565) If the minus and second *letter* appear, the second letter shall follow the first letter alphabetically.

C593 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 *letter*s 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.

4 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 5.37

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 5.38

The following is an example of a mapping to a derived type that is inaccessible in the local scope:

```
     PROGRAM MAIN
       IMPLICIT TYPE(BLOB) (A)
       TYPE BLOB
         INTEGER :: I
       END TYPE BLOB
       TYPE(BLOB) :: B
```

NOTE 5.38 (cont.)

```
    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 5.39

Implicit typing is not affected by BLOCK constructs. For example, in

```
  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`.

5.8 NAMELIST statement

1 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 (9.6, 10.11).

R567 *namelist-stmt* **is** NAMELIST ■
■ / *namelist-group-name* / *namelist-group-object-list* ■
■ [[,] / *namelist-group-name* / ■
■ *namelist-group-object-list*] ...

C594 (R567) The *namelist-group-name* shall not be a name accessed by use association.

R568 *namelist-group-object* **is** *variable-name*

C595 (R568) A *namelist-group-object* shall not be an assumed-size array.

C596 (R567) A *namelist-group-object* shall not have the PRIVATE attribute if the *namelist-group-name* has the PUBLIC attribute.

2 The order in which the variables are specified in the NAMELIST statement determines the order in which the values appear on output.

3 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*.

4 A namelist group object may be a member of more than one namelist group.

5 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 5.40

An example of a NAMELIST statement is:

```
NAMELIST /NLIST/ A, B, C
```

5.9 Storage association of data objects

5.9.1 EQUIVALENCE statement

5.9.1.1 General

1 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 (16.5.3) of the objects that share the storage units.

2 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.

R569	*equivalence-stmt*	**is**	EQUIVALENCE *equivalence-set-list*
R570	*equivalence-set*	**is**	(*equivalence-object* , *equivalence-object-list*)
R571	*equivalence-object*	**is**	*variable-name*
		or	*array-element*
		or	*substring*

C597 (R571) 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 object, a coarray, a variable with the BIND attribute, a variable in a common block that has the BIND attribute, or a named constant.

C598 (R571) An *equivalence-object* shall not be a designator that has more than one *part-ref*.

C599 (R571) An *equivalence-object* shall not have the TARGET attribute.

C5100 (R571) Each subscript or substring range expression in an *equivalence-object* shall be an integer constant expression (7.1.12).

C5101 (R570) If an *equivalence-object* is default integer, default real, double precision real, default complex, default logical, or of numeric sequence type, all of the objects in the equivalence set shall be of these types and kinds.

C5102 (R570) If an *equivalence-object* is default character or of character sequence type, all of the objects in the equivalence set shall be of these types and kinds.

C5103 (R570) If an *equivalence-object* is of a sequence type that is not a numeric sequence or character sequence type, all of the objects in the equivalence set shall be of the same type with the same type parameter values.

C5104 (R570) If an *equivalence-object* is of an intrinsic type but is not default integer, default real, double precision real, default complex, default logical, or default character, all of the objects in the equivalence set shall be of the same type with the same kind type parameter value.

C5105 (R571) If an *equivalence-object* has the PROTECTED attribute, all of the objects in the equivalence set shall have the PROTECTED attribute.

C5106 (R571) The name of an *equivalence-object* shall not be a name made accessible by use association.

C5107 (R571) A *substring* shall not have length zero.

NOTE 5.41

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.

A structure that appears in an EQUIVALENCE statement shall be a sequence structure. If a sequence structure is not of numeric sequence type or of character sequence type, it shall be equivalenced only to objects of the same type with the same type parameter values.

A structure of a numeric sequence type shall be equivalenced only to another structure of a numeric sequence type, an object that is default integer, default real, double precision real, default complex, or default logical type such that components of the structure ultimately become associated only with objects of these types and kinds.

A structure of a character sequence type shall be equivalenced only to a default character object or another structure of a character sequence type.

An object of intrinsic type with nondefault kind type parameters shall not be equivalenced to objects of different type or kind type parameters.

Further rules on the interaction of EQUIVALENCE statements and default initialization are given in 16.5.3.4.

5.9.1.2 Equivalence association

1 An EQUIVALENCE statement specifies that the storage sequences (16.5.3.2) 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.

2 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.

5.9.1.3 Equivalence of default character objects

1 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.

2 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 5.42

For example, using the declarations:

```
CHARACTER (LEN = 4) :: A, B
CHARACTER (LEN = 3) :: C (2)
EQUIVALENCE (A, C (1)), (B, C (2))
```

the association of A, B, and C can be illustrated graphically as:

```
  1      2      3      4      5      6      7
 |---    --- A  ---    ---|
                      |---    --- B  ---    ---|
 |---    C(1)   ---|  |---    C(2)   ---|
```

5.9.1.4 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.

5.9.1.5 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.

5.9.2 COMMON statement

5.9.2.1 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 (16.5.3).

2 A common block that does not have a name is called blank common.

R572	*common-stmt*	**is**	COMMON ■ ■ [/ [*common-block-name*] /] *common-block-object-list* ■ ■ [[,] / [*common-block-name*] / ■ ■ *common-block-object-list*] ...
R573	*common-block-object*	**is**	*variable-name* [(*array-spec*)]

C5108 (R573) An *array-spec* in a *common-block-object* shall be an *explicit-shape-spec-list*.

C5109 (R573) Only one appearance of a given *variable-name* is permitted in all *common-block-object-list*s within a scoping unit.

C5110 (R573) 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 object, a variable with the BIND attribute, an unlimited polymorphic pointer, or a coarray.

C5111 (R573) 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.

C5112 (R573) 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 between them declares the data objects whose names appear in the common block object list that follows to be in 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 (4.5.2.3) appear in common, it is as if the individual components were enumerated directly in the common list.

5.9.2.2 Common block storage sequence

1 For each common block in a scoping unit, a common block storage sequence is formed as follows:

(1) A storage sequence is formed consisting of the sequence of storage units in the storage sequences (16.5.3.2) 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.

(2) 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.

2 Only COMMON statements and EQUIVALENCE statements appearing in the scoping unit contribute to common block storage sequences formed in that scoping unit.

5.9.2.3 Size of a common block

1 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.

5.9.2.4 Common association

1 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.

2 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.

3 A nonpointer object that is default character or of character sequence type shall be associated only with nonpointer objects of these types and kinds.

4 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.

5 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.

6 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.

7 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 5.43

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 (5.9.1).

5.9.2.5 Differences between named common and blank common

1 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 (16.6.6).
- 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 (11.3), but objects in blank common shall not be initially defined.

5.9.3 Restrictions on common and equivalence

1 An EQUIVALENCE statement shall not cause the storage sequences of two different common blocks to be associated.

2 Equivalence association shall not cause a derived-type object with default initialization to be associated with an object in a common block.

3 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.

6 Use of data objects

6.1 Designator

R601 *designator* **is** *object-name*
or *array-element*
or *array-section*
or *coindexed-named-object*
or *complex-part-designator*
or *structure-component*
or *substring*

1 The appearance of a data object designator in a context that requires its value is termed a reference.

6.2 Variable

R602 *variable* **is** *designator*
or *function-reference*

C601 (R602) *designator* shall not be a constant or a subobject of a constant.

C602 (R602) *function-reference* shall have a data pointer result.

1 A variable is either the data object denoted by *designator* or the target of *expr*.

2 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 data object becomes defined with a value when events described in 16.6.5 occur.

R603 *variable-name* **is** *name*

C603 (R603) *variable-name* shall be the name of a variable.

R604 *logical-variable* **is** *variable*

C604 (R604) *logical-variable* shall be of type logical.

R605 *char-variable* **is** *variable*

C605 (R605) *char-variable* shall be of type character.

R606 *default-char-variable* **is** *variable*

C606 (R606) *default-char-variable* shall be default character.

R607 *int-variable* **is** *variable*

C607 (R607) *int-variable* shall be of type integer.

NOTE 6.1

For example, given the declarations:

```
CHARACTER (10)  A, B (10)
TYPE (PERSON)  P   ! See Note 4.16
```

NOTE 6.1 (cont.)

then A, B, B (1), B (1:5), P % AGE, and A (1:1) are all variables.

6.3 Constants

1 A constant (3.2.3) 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 (5.5.13, 5.6.11). A reference to a constant is always permitted; redefinition of a constant is never permitted.

6.4 Scalars

6.4.1 Substrings

1 A substring is a contiguous portion of a character string (4.4.4).

R608 *substring* **is** *parent-string* (*substring-range*)

R609 *parent-string* **is** *scalar-variable-name*
or *array-element*
or *coindexed-named-object*
or *scalar-structure-component*
or *scalar-constant*

R610 *substring-range* **is** [*scalar-int-expr*] : [*scalar-int-expr*]

C608 (R609) *parent-string* shall be of type character.

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 6.2

Examples of character substrings are:

```
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
```

6.4.2 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.

R611 *data-ref* **is** *part-ref* [% *part-ref*] ...

R612 *part-ref* **is** *part-name* [(*section-subscript-list*)] [*image-selector*]

C609 (R611) Each *part-name* except the rightmost shall be of derived type.

C610 (R611) Each *part-name* except the leftmost shall be the name of a component of the declared type of the preceding *part-name*.

C611 (R611) If the rightmost *part-name* is of abstract type, *data-ref* shall be polymorphic.

C612 (R611) The leftmost *part-name* shall be the name of a data object.

C613 (R612) If a *section-subscript-list* appears, the number of *section-subscript*s shall equal the rank of *part-name*.

C614 (R612) If *image-selector* appears, the number of *cosubscript*s shall be equal to the corank of *part-name*.

C615 (R612) If *image-selector* appears and *part-name* is an array, *section-subscript-list* shall appear.

C616 (R611) If *image-selector* appears, *data-ref* shall not be of type C_PTR or C_FUNPTR (15.3.3).

C617 (R611) 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 subcomponent.

2 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.

C618 (R611) 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.

3 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.

4 The type and type parameters, if any, of a *data-ref* are those of the rightmost part name.

5 A *data-ref* with more than one *part-ref* is a subobject of its base object if none of the *part-name*s, except for possibly the rightmost, are pointers. 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 6.3

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.

R613 *structure-component* **is** *data-ref*

C619 (R613) There shall be more than one *part-ref* and the rightmost *part-ref* shall not have a *section-subscript-list*.

6 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 is defined to have the POINTER attribute.

NOTE 6.4

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 6.4 (cont.)

> For a more elaborate example see C.3.1.

NOTE 6.5

> 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.

6.4.3 Coindexed named objects

1 A *coindexed-named-object* is a named scalar coarray variable followed by an image selector.

R614 *coindexed-named-object* **is** *data-ref*

C620 (R614) 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.

6.4.4 Complex parts

R615 *complex-part-designator* **is** *designator* % RE
or *designator* % IM

C621 (R615) 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*.

NOTE 6.6

The following are examples of complex part designators:

```
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
```

6.4.5 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.

R616 *type-param-inquiry* **is** *designator* % *type-param-name*

C622 (R616) 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 6.7

> 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 6.7 (cont.)

The intrinsic type parameters can also be inquired about by using the intrinsic functions KIND and LEN.

NOTE 6.8

The following are examples of type parameter inquiries:

```
a%kind       !-- A is real.  Same value as KIND(a).
s%len        !-- S is character.  Same value as LEN(s).
b(10)%kind   !-- Inquiry about an array element.
p%dim        !-- P is of the derived type general_point.
```

See Note 4.23 for the definition of the `general_point` type used in the last example above.

6.5 Arrays

6.5.1 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 (6.5.3.2) is specified.

6.5.2 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 (2.4.6). 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 (5.9.1.4). An assumed-size array (5.5.8.5) 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.

6.5.3 Array elements and array sections

6.5.3.1 Syntax

R617 *array-element* **is** *data-ref*

C623 (R617) Every *part-ref* shall have rank zero and the last *part-ref* shall contain a *subscript-list*.

R618 *array-section* **is** *data-ref* [(*substring-range*)]
or *complex-part-designator*

C624 (R618) 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.

C625 (R618) If a *substring-range* appears, the rightmost *part-name* shall be of type character.

R619 *subscript* **is** *scalar-int-expr*

R620 *section-subscript* **is** *subscript*
or *subscript-triplet*
or *vector-subscript*

R621 *subscript-triplet* **is** [*subscript*] : [*subscript*] [: *stride*]

R622 *stride* **is** *scalar-int-expr*

R623 *vector-subscript* **is** *int-expr*

C626 (R623) A *vector-subscript* shall be an integer array expression of rank one.

C627 (R621) The second subscript shall not be omitted from a *subscript-triplet* in the last dimension of an assumed-size array.

1 An array element is a scalar. An array section is an array. If a *substring-range* appears in an *array-section*, each element is the designated substring of the corresponding element of the array section.

2 The value of a subscript in an array element shall be within the bounds for its dimension.

NOTE 6.9

For example, with the declarations:

```
REAL A (10, 10)
CHARACTER (LEN = 10) B (5, 5, 5)
```

A (1, 2) is an array element, A (1:N:2, M) is a rank-one array section, and B (:, :, :) (2:3) is an array of shape (5, 5, 5) whose elements are substrings of length 2 of the corresponding elements of B.

NOTE 6.10

Unless otherwise specified, an array element or array section does not have an attribute of the whole array. In particular, an array element or an array section does not have the POINTER or ALLOCATABLE attribute.

NOTE 6.11

Examples of array elements and array sections are:

```
    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
```

6.5.3.2 Array element order

1 The elements of an array form a sequence known as the array element order. The position of an array element in this sequence is determined by the subscript order value of the subscript list designating the element. The subscript order value is computed from the formulas in Table 6.1.

Table 6.1: **Subscript order value**

Rank	Subscript bounds	Subscript list	Subscript order value
1	$j_1{:}k_1$	s_1	$1 + (s_1 - j_1)$
2	$j_1{:}k_1$,$j_2{:}k_2$	s_1, s_2	$1 + (s_1 - j_1)$ $+(s_2 - j_2) \times d_1$
3	$j_1{:}k_1, j_2{:}k_2, j_3{:}k_3$	s_1, s_2, s_3	$1 + (s_1 - j_1)$ $+(s_2 - j_2) \times d_1$ $+(s_3 - j_3) \times d_2 \times d_1$
. . .	. . .	. . .	. . .

Subscript order value (cont.)

Rank	Subscript bounds	Subscript list	Subscript order value
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 6.1: 1) $d_i = \max(k_i - j_i + 1, 0)$ is the size of the ith dimension. 2) If the size of the array is nonzero, $j_i \leq s_i \leq k_i$ for all i = 1, 2, ..., 15.			

6.5.3.3 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 ith element is the number of integer values in the sequence indicated by the ith 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.

6.5.3.3.1 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 6.12

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):

```
A (3, 2, 1)     A (3, 2, 2)
A (4, 2, 1)     A (4, 2, 2)
A (5, 2, 1)     A (5, 2, 2)
```

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 6.13

> 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 6.14

> 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.

6.5.3.3.2 Vector subscript

1 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.

2 An array section with a vector subscript shall not be finalized by a nonelemental final subroutine.

3 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 6.15

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.

6.5.4 Simply contiguous array designators

1 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*.

2 An array designator is simply contiguous if and only if it is

- an *object-name* that has the CONTIGUOUS attribute,

- an *object-name* that is not a pointer, not assumed-shape, and not assumed-rank,
- a *structure-component* whose final *part-name* is an array and that either has the CONTIGUOUS attribute or is not a pointer, or
- an array section
  - that is not a *complex-part-designator*,
  - that does not have a *substring-range*,
  - whose final *part-ref* has nonzero rank,
  - whose rightmost *part-name* has the CONTIGUOUS attribute or is neither assumed-shape nor a pointer, and
  - which either does not have a *section-subscript-list*, or has a *section-subscript-list* which specifies a simply contiguous section.

3 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 6.16

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.

6.6 Image selectors

1 An image selector determines the image index for a coindexed object.

R624 *image-selector* **is** *lbracket cosubscript-list rbracket*

R625 *cosubscript* **is** *scalar-int-expr*

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 (6.5.3.2), taking account of the bounds. An image selector shall specify an image index value that is not greater than the number of images.

NOTE 6.17

For example, if there are 16 images and the coarray A is declared

```
REAL :: A(10)[5,*]
```

A(:)[1,4] is valid because it specifies image 16, but A(:)[2,4] is invalid because it specifies image 17.

6.7 Dynamic association

6.7.1 ALLOCATE statement

6.7.1.1 Form of the ALLOCATE statement

1 The ALLOCATE statement dynamically creates pointer targets and allocatable variables.

R626 *allocate-stmt* **is** ALLOCATE ([*type-spec* ::] *allocation-list* ■
■ [, *alloc-opt-list*])

R627 *alloc-opt* **is** ERRMSG = *errmsg-variable*
or MOLD = *source-expr*
or SOURCE = *source-expr*
or STAT = *stat-variable*

R628 *stat-variable* **is** *scalar-int-variable*

R629 *errmsg-variable* **is** *scalar-default-char-variable*

R630 *source-expr* **is** *expr*

R631 *allocation* **is** *allocate-object* [(*allocate-shape-spec-list*)] ■
■ [*lbracket allocate-coarray-spec rbracket*]

R632 *allocate-object* **is** *variable-name*
or *structure-component*

R633 *allocate-shape-spec* **is** [*lower-bound-expr* :] *upper-bound-expr*

R634 *lower-bound-expr* **is** *scalar-int-expr*

R635 *upper-bound-expr* **is** *scalar-int-expr*

R636 *allocate-coarray-spec* **is** [*allocate-coshape-spec-list* ,] [*lower-bound-expr* :] *

R637 *allocate-coshape-spec* **is** [*lower-bound-expr* :] *upper-bound-expr*

C628 (R632) Each *allocate-object* shall be a data pointer or an allocatable variable.

C629 (R626) 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.

C630 (R626) If *type-spec* appears, it shall specify a type with which each *allocate-object* is type compatible.

C631 (R626) 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.

C632 (R626) 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*.

C633 (R626) 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*.

C634 (R631) If *allocate-object* is scalar, *allocate-shape-spec-list* shall not appear.

C635 (R631) An *allocate-coarray-spec* shall appear if and only if the *allocate-object* is a coarray.

C636 (R631) The number of *allocate-shape-spec*s in an *allocate-shape-spec-list* shall be the same as the rank of the *allocate-object*. The number of *allocate-coshape-spec*s in an *allocate-coarray-spec* shall be one less than the corank of the *allocate-object*.

C637 (R627) No *alloc-opt* shall appear more than once in a given *alloc-opt-list*.

C638 (R626) At most one of *source-expr* and *type-spec* shall appear.

C639 (R626) Each *allocate-object* shall be type compatible (4.3.2.3) with *source-expr*. If SOURCE= appears, *source-expr* shall be a scalar or have the same rank as each *allocate-object*.

C640 (R626) 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*.

C641 (R626) *type-spec* shall not specify a type that has a coarray ultimate component.

C642 (R626) *type-spec* shall not specify the type C_PTR or C_FUNPTR if an *allocate-object* is a coarray.

C643 (R626) The declared type of *source-expr* shall not be C_PTR, C_FUNPTR, LOCK_TYPE, or have a subcomponent of type LOCK_TYPE, if an *allocate-object* is a coarray.

C644 (R630) The declared type of *source-expr* shall not have a coarray ultimate component.

C645 (R632) An *allocate-object* shall not be a coindexed object.

NOTE 6.18

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[:]
...
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-object* is a coarray, the ALLOCATE statement shall not have a *source-expr* with a dynamic type of C_PTR, C_FUNPTR, or LOCK_TYPE, or which has a subcomponent whose dynamic type is 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 *type-spec* appears and the value of a 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.

7 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 6.19

An example of an ALLOCATE statement is:

```
ALLOCATE (X (N), B (-3 : M, 0:9), STAT = IERR_ALLOC)
```

6.7.1.2 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 image. The values of corresponding bounds and corresponding cobounds shall be the same on every image. If the coarray is a dummy argument, its ultimate argument (12.5.2.3) shall be the same coarray on every image.

4 When an ALLOCATE statement is executed for which an *allocate-object* is a coarray, there is an implicit synchronization of all images. On each image, execution of the segment (8.5.2) following the statement is delayed until all other images have executed the same statement the same number of times.

NOTE 6.20

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).

5 If *source-expr* is a pointer, it shall be associated with a target. If *source-expr* is allocatable, it shall be allocated.

6 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.

7 If SOURCE= appears, *source-expr* shall be conformable with *allocation*. If the value of a nondeferred length type parameter of *allocate-object* is different from the value of the corresponding type parameter of *source-expr*, an error condition occurs. 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.

8 If MOLD= appears and *source-expr* is a variable, its value need not be defined.

9 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 6.7.4. The ERRMSG= specifier is described in 6.7.5.

6.7.1.3 Allocation of allocatable variables

1 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 (13.7.119). An allocatable variable with this status may be referenced, defined, or deallocated; allocating it causes an error condition in the ALLOCATE statement. The intrinsic function ALLOCATED (13.7.11) returns 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 (6.7.3) or if it is given that status by the allocation transfer procedure. 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 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 intrinsic function ALLOCATED (13.7.11) returns false for such a variable.

2 At the beginning of execution of a program, allocatable variables are unallocated.

3 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.

4 An unsaved allocatable local variable of a procedure has a status of unallocated at the beginning of each invocation of the procedure. An unsaved local variable of a construct has a status of unallocated at the beginning of each execution of the construct.

5 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.

6 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.

6.7.1.4 Allocation of pointer targets

1 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 (13.7.16) may be used to determine whether a pointer that does not have undefined association status is associated.

2 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.

6.7.2 NULLIFY statement

1 The NULLIFY statement causes pointers to be disassociated.

R638 *nullify-stmt* **is** NULLIFY (*pointer-object-list*)

R639 *pointer-object* **is** *variable-name*
or *structure-component*
or *proc-pointer-name*

C646 (R639) Each *pointer-object* shall have the POINTER attribute.

2 A *pointer-object* shall not depend on the value, bounds, or association status of another *pointer-object* in the same NULLIFY statement.

NOTE 6.21

When a NULLIFY statement is applied to a polymorphic pointer (4.3.2.3), its dynamic type becomes the declared type.

6.7.3 DEALLOCATE statement

6.7.3.1 Form of the DEALLOCATE statement

1 The DEALLOCATE statement causes allocatable variables to be deallocated; it causes pointer targets to be deallocated and the pointers to be disassociated.

R640 *deallocate-stmt* **is** DEALLOCATE (*allocate-object-list* [, *dealloc-opt-list*])

R641 *dealloc-opt* **is** STAT = *stat-variable*
or ERRMSG = *errmsg-variable*

C647 (R641) No *dealloc-opt* shall appear more than once in a given *dealloc-opt-list*.

2 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.

3 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 6.7.4. The ERRMSG= specifier is described in 6.7.5.

4 When more than one allocated object is deallocated by execution of a DEALLOCATE statement, the order of deallocation is processor dependent.

NOTE 6.22

An example of a DEALLOCATE statement is:

```
DEALLOCATE (X, B)
```

6.7.3.2 Deallocation of allocatable variables

1 Deallocating an unallocated allocatable variable causes an error condition in the DEALLOCATE statement. Deallocating 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.

2 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.

3 When a BLOCK construct terminates, any unsaved allocated allocatable local variable of the construct is deallocated.

4 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.

5 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.

6 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).

7 When an intrinsic assignment statement (7.2.1.3) is executed, any noncoarray allocated allocatable subobject of the variable is deallocated before the assignment takes place.

8 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 deallocated.

9 If an allocatable component is a subobject of a finalizable object, that object is finalized before the component is automatically deallocated.

10 When a statement that deallocates a coarray is executed, there is an implicit synchronization of all images. On each image, execution of the segment (8.5.2) following the statement is delayed until all other images have executed the same statement the same number of times. If an *allocate-object* is a coarray dummy argument, its ultimate argument (12.5.2.3) shall be the same coarray on every image.

11 The effect of automatic deallocation is the same as that of a DEALLOCATE statement without a *dealloc-opt-list*.

NOTE 6.23

In the following example:

```
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 6.24

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.

6.7.3.3 Deallocation of pointer targets

1 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.

2 If a pointer appears in a DEALLOCATE statement, it shall be associated with the whole of an object that was created by allocation. The pointer shall have the same dynamic type and type parameters as the allocated object, and if the allocated object is an array the pointer shall be an array whose elements are the same as those of the allocated object in array element order. Deallocating a pointer target causes the pointer association status of any other pointer that is associated with the target or a portion of the target to become undefined.

6.7.4 STAT= specifier

1 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.

2 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.

3 If an ALLOCATE or DEALLOCATE statement with a coarray *allocate-object* is executed when one or more images has initiated 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 (13.8.2). 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. In either 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 6.25

The status of objects that were not successfully allocated or deallocated can be individually checked with the intrinsic functions ALLOCATED or ASSOCIATED.

6.7.5 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.

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.

7 Expressions and assignment

7.1 Expressions

7.1.1 Expression semantics

1 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 (7.1.9). The corank of an expression that is not a variable is zero.

7.1.2 Form of an expression

7.1.2.1 Overall expression syntax

1 An expression is formed from operands, operators, and parentheses. An operand is either a scalar or an array. An operation is either intrinsic (7.1.5) or defined (7.1.6). More complicated expressions can be formed using operands which are themselves expressions.

2 An expression is defined in terms of several categories: primary, level-1 expression, level-2 expression, level-3 expression, level-4 expression, and level-5 expression.

3 These categories are related to the different operator precedence levels and, in general, are defined in terms of other categories. The simplest form of each expression category is a *primary*.

7.1.2.2 Primary

R701	*primary*	**is**	*constant*
		or	*designator*
		or	*array-constructor*
		or	*structure-constructor*
		or	*function-reference*
		or	*type-param-inquiry*
		or	*type-param-name*
		or	(*expr*)

C701 (R701) The *type-param-name* shall be the name of a type parameter.

C702 (R701) The *designator* shall not be a whole assumed-size array.

C703 (R701) The *expr* shall not be a function reference that returns a procedure pointer.

NOTE 7.1

Examples of a *primary* are:

Example	Syntactic class
`1.0`	*constant*
`'ABCDEFGHIJKLMNOPQRSTUVWXYZ' (I:I)`	*designator*
`[ 1.0, 2.0 ]`	*array-constructor*
`PERSON (12, 'Jones')`	*structure-constructor*
`F (X, Y)`	*function-reference*
`X%KIND`	*type-param-inquiry*

NOTE 7.1 (cont.)

`KIND`	*type-param-name*
`(S + T)`	(*expr*)

7.1.2.3 Level-1 expressions

1 Defined unary operators have the highest operator precedence (Table 7.1). Level-1 expressions are primaries optionally operated on by defined unary operators:

R702 *level-1-expr* **is** [*defined-unary-op*] *primary*

R703 *defined-unary-op* **is** . *letter* [*letter*] ... .

C704 (R703) 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 7.2

Simple examples of a level-1 expression are:

Example	Syntactic class
`A`	*primary* (R701)
`.INVERSE. B`	*level-1-expr* (R702)

A more complicated example of a level-1 expression is:

```
.INVERSE. (A + B)
```

7.1.2.4 Level-2 expressions

1 Level-2 expressions are level-1 expressions optionally involving the numeric operators *power-op*, *mult-op*, and *add-op*.

R704 *mult-operand* **is** *level-1-expr* [*power-op mult-operand*]

R705 *add-operand* **is** [*add-operand mult-op*] *mult-operand*

R706 *level-2-expr* **is** [[*level-2-expr*] *add-op*] *add-operand*

R707 *power-op* **is** **

R708 *mult-op* **is** *
or /

R709 *add-op* **is** +
or −

NOTE 7.3

Simple examples of a level-2 expression are:

Example	Syntactic class	Remarks
`A`	*level-1-expr*	A is a *primary*. (R702)
`B ** C`	*mult-operand*	B is a *level-1-expr*, ** is a *power-op*, and C is a *mult-operand*. (R704)
`D * E`	*add-operand*	D is an *add-operand*, * is a *mult-op*, and E is a *mult-operand*. (R705)
`+1`	*level-2-expr*	+ is an *add-op* and 1 is an *add-operand*. (R706)
`F - I`	*level-2-expr*	F is a *level-2-expr*, − is an *add-op*, and I is an *add-operand*. (R706)

NOTE 7.3 (cont.)

A more complicated example of a level-2 expression is:

```
- A + D * E + B ** C
```

7.1.2.5 Level-3 expressions

1 Level-3 expressions are level-2 expressions optionally involving the character operator *concat-op*.

R710 *level-3-expr* **is** [*level-3-expr concat-op*] *level-2-expr*

R711 *concat-op* **is** //

NOTE 7.4

Simple examples of a level-3 expression are:

Example	Syntactic class
`A`	*level-2-expr* (R706)
`B // C`	*level-3-expr* (R710)

A more complicated example of a level-3 expression is:

```
X // Y // 'ABCD'
```

7.1.2.6 Level-4 expressions

1 Level-4 expressions are level-3 expressions optionally involving the relational operators *rel-op*.

R712 *level-4-expr* **is** [*level-3-expr rel-op*] *level-3-expr*

R713 *rel-op* **is** .EQ.
or .NE.
or .LT.
or .LE.
or .GT.
or .GE.
or ==
or /=
or <
or <=
or >
or >=

NOTE 7.5

Simple examples of a level-4 expression are:

Example	Syntactic class
`A`	*level-3-expr* (R710)
`B == C`	*level-4-expr* (R712)
`D < E`	*level-4-expr* (R712)

A more complicated example of a level-4 expression is:

```
(A + B) /= C
```

7.1.2.7 Level-5 expressions

1 Level-5 expressions are level-4 expressions optionally involving the logical operators *not-op*, *and-op*, *or-op*, and *equiv-op*.

R714 *and-operand* **is** [*not-op*] *level-4-expr*

R715 *or-operand* **is** [*or-operand and-op*] *and-operand*

R716 *equiv-operand* **is** [*equiv-operand or-op*] *or-operand*

R717 *level-5-expr* **is** [*level-5-expr equiv-op*] *equiv-operand*

R718 *not-op* **is** .NOT.

R719 *and-op* **is** .AND.

R720 *or-op* **is** .OR.

R721 *equiv-op* **is** .EQV.
or .NEQV.

NOTE 7.6

Simple examples of a level-5 expression are:

Example	Syntactic class
`A`	*level-4-expr* (R712)
`.NOT. B`	*and-operand* (R714)
`C .AND. D`	*or-operand* (R715)
`E .OR. F`	*equiv-operand* (R716)
`G .EQV. H`	*level-5-expr* (R717)
`S .NEQV. T`	*level-5-expr* (R717)

A more complicated example of a level-5 expression is:

```
A .AND. B .EQV. .NOT. C
```

7.1.2.8 General form of an expression

1 Expressions are level-5 expressions optionally involving defined binary operators. Defined binary operators have the lowest operator precedence (Table 7.1).

R722 *expr* **is** [*expr defined-binary-op*] *level-5-expr*

R723 *defined-binary-op* **is** . *letter* [*letter*] ... .

C705 (R723) 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 7.7

Simple examples of an expression are:

Example	Syntactic class
`A`	*level-5-expr* (R717)
`B.UNION.C`	*expr* (R722)

More complicated examples of an expression are:

NOTE 7.7 (cont.)

```
     (B .INTERSECT. C) .UNION. (X - Y)
A + B == C * D
.INVERSE. (A + B)
A + B .AND. C * D
E // G == H (1:10)
```

7.1.3 Precedence of operators

1 There is a precedence among the intrinsic and extension operations corresponding to the form of expressions specified in 7.1.2, 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 7.1.

Table 7.1: **Categories of operations and relative precedence**

Category of operation	Operators	Precedence
Extension	*defined-unary-op*	Highest
Numeric	**	.
Numeric	*, /	.
Numeric	unary +, −	.
Numeric	binary +, −	.
Character	//	.
Relational	.EQ., .NE., .LT., .LE., .GT., .GE., ==, /=, <, <=, >, >=	.
Logical	.NOT.	.
Logical	.AND.	.
Logical	.OR.	.
Logical	.EQV., .NEQV.	.
Extension	*defined-binary-op*	Lowest

2 The precedence of a defined operation is that of its operator.

NOTE 7.8

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)
```

3 The general form of an expression (7.1.2) 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 7.9

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.

For example, the expressions

NOTE 7.9 (cont.)

```
2.1 + 3.4 + 4.9
2.1 * 3.4 * 4.9
2.1 / 3.4 / 4.9
2 ** 3 ** 4
'AB' // 'CD' // 'EF'
```

have the same interpretations as the expressions

```
(2.1 + 3.4) + 4.9
(2.1 * 3.4) * 4.9
(2.1 / 3.4) / 4.9
2 ** (3 ** 4)
('AB' // 'CD') // 'EF'
```

As a consequence of the general form (7.1.2), 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:

```
A * .INVERSE. B
- .INVERSE. (B)
```

As another example, in the expression

```
A .OR. B .AND. C
```

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

```
A .OR. (B .AND. C)
```

NOTE 7.10

An expression can contain more than one category of operator. The logical expression

```
L .OR. A + B >= C
```

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

```
L .OR. ((A + B) >= C)
```

NOTE 7.11

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.

7.1.4 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 (8.1.7.4), the mask expression in a WHERE statement (7.2.3.1), or the *concurrent-limit*s and *concurrent-step*s in a FORALL statement (7.2.4) 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 7.12

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

```
IF (F (X)) A = X
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 7.13

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 (12.8), the element operations may be performed simultaneously or in any order.

7.1.5 Intrinsic operations

7.1.5.1 Intrinsic operation classification

1 An intrinsic operation is either a unary or binary operation. An intrinsic unary operation is an operation of the form *intrinsic-operator* x_2 where x_2 is of an intrinsic type (4.4) listed in Table 7.2 for the unary intrinsic operator.

2 An intrinsic binary operation is an operation of the form x_1 *intrinsic-operator* x_2 where x_1 and x_2 are conformable and of the intrinsic types (4.4) listed in Table 7.2 for the binary intrinsic operator.

3 A numeric intrinsic operation is an intrinsic operation for which the *intrinsic-operator* is a numeric operator (+, −, *, /, or **). A numeric intrinsic operator is the operator in a numeric intrinsic operation.

4 The character intrinsic operation is the intrinsic operation for which the *intrinsic-operator* is (//) and both operands are of type character with the same kind type parameter. The character intrinsic operator is the operator in a character intrinsic operation.

5 A logical intrinsic operation is an intrinsic operation for which the *intrinsic-operator* is .AND., .OR., .NOT., .EQV., or .NEQV. and both operands are of type logical. A logical intrinsic operator is the operator in a logical intrinsic operation.

6 A relational intrinsic operator is an *intrinsic-operator* that is .EQ., .NE., .GT., .GE., .LT., .LE., ==, /=, >, >=, <, or <=. A relational intrinsic operation is an intrinsic operation for which the *intrinsic-operator* is a relational intrinsic operator. A numeric relational intrinsic operation is a relational intrinsic operation for which both operands are of numeric type. A character relational intrinsic operation is a relational intrinsic operation for which both operands are of type character. The kind type parameters of the operands of a character relational intrinsic operation shall be the same.

7 The interpretations defined in subclause 7.1.5 apply to both scalars and arrays; the interpretation for arrays is obtained by applying the interpretation for scalars element by element.

NOTE 7.14

For example, if X is of type real, J is of type integer, and INT is the real-to-integer intrinsic conversion function, the expression INT (X + J) is an integer expression and X + J is a real expression.

Table 7.2: **Type of operands and results for intrinsic operators**

Intrinsic operator *op*	Type of x_1	Type of x_2	Type of $[x_1]$ *op* x_2
Unary +, −		I, R, Z	I, R, Z
Binary +, −, *, /, **	I	I, R, Z	I, R, Z
	R	I, R, Z	R, R, Z
	Z	I, R, Z	Z, Z, Z
//	C	C	C
.EQ., .NE., ==, /=	I	I, R, Z	L, L, L
	R	I, R, Z	L, L, L
	Z	I, R, Z	L, L, L
	C	C	L
.GT., .GE., .LT., .LE. >, >=, <, <=	I	I, R	L, L
	R	I, R	L, L
	C	C	L
.NOT.		L	L
.AND., .OR., .EQV., .NEQV.	L	L	L

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.

7.1.5.2 Numeric intrinsic operations

7.1.5.2.1 Interpretation of numeric intrinsic operations

1 The two operands of numeric intrinsic binary operations may be of different numeric types or different kind type parameters. Except for a value raised to an integer power, if the operands have different types or kind type parameters, the effect is as if each operand that differs in type or kind type parameter from those of the result is converted to the type and kind 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.

2 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 7.1.5.1.

3 The numeric operators and their interpretation in an expression are given in Table 7.3, 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 7.3: **Interpretation of the numeric intrinsic operators**

Operator	Representing	Use of operator	Interpretation
**	Exponentiation	x_1 ** x_2	Raise x_1 to the power x_2
/	Division	x_1 / x_2	Divide x_1 by x_2
*	Multiplication	x_1 * x_2	Multiply x_1 by x_2
−	Subtraction	x_1 - x_2	Subtract x_2 from x_1
−	Negation	- x_2	Negate x_2
+	Addition	x_1 + x_2	Add x_1 and x_2
+	Identity	+ x_2	Same as x_2

4 The interpretation of a division operation depends on the types of the operands (7.1.5.2.2).

5 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 $1/(x_1$ ** ABS $(x_2))$, which is subject to the rules of integer division (7.1.5.2.2).

NOTE 7.15

For example, 2 ** (-3) has the value of $1/(2$ ** $3)$, which is zero.

7.1.5.2.2 Integer division

1 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 7.2 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 7.16

For example, the expression (-8) / 3 has the value (-2).

7.1.5.2.3 Complex exponentiation

1 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 $x_1^{x_2}$.

7.1.5.2.4 Evaluation of numeric intrinsic operations

1 The execution of any numeric operation whose result is not defined by the arithmetic used by the processor is prohibited. Raising a negative-valued primary of type real to a real power is prohibited.

2 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.

3 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 7.17

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 7.1.5.2.2.

NOTE 7.18

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.

Expression	Allowable alternative form
X + Y	Y + X
X * Y	Y * X
-X + Y	Y - X
X + Y + Z	X + (Y + Z)
X - Y + Z	X - (Y - Z)
X * A / Z	X * (A / Z)
X * Y - X * Z	X * (Y - Z)
A / B / C	A / (B * C)
A / 5.0	0.2 * A

The following are examples of expressions with forbidden alternative forms that cannot be used by a processor in the evaluation of those expressions.

Expression	Forbidden alternative form
I / 2	0.5 * I
X * I / J	X * (I / J)
I / J / A	I / (J * A)
(X + Y) + Z	X + (Y + Z)
(X * Y) - (X * Z)	X * (Y - Z)
X * (Y - Z)	X * Y - X * Z

NOTE 7.19

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 / J
A * (I / J)
```

could have different mathematical values if I and J are of type integer.

NOTE 7.20

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.

7.1.5.3 Character intrinsic operation

7.1.5.3.1 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 7.4, 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 7.4: **Interpretation of the character intrinsic operator //**

Operator	Representing	Use of operator	Interpretation
//	Concatenation	x_1 // x_2	Concatenate x_1 with x_2

3 The result of the character intrinsic operation // 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 7.21

For example, the value of ('AB' // 'CDE') // 'F' is the string 'ABCDEF'. Also, the value of 'AB' // ('CDE' // 'F') is the string 'ABCDEF'.

7.1.5.3.2 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 7.22

For example, the statements

```
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.

7.1.5.4 Logical intrinsic operations

7.1.5.4.1 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 7.1.5.1.

2 The logical operators and their interpretation when used to form an expression are given in Table 7.5, 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 7.5: **Interpretation of the logical intrinsic operators**

Operator	Representing	Use of operator	Interpretation
.NOT.	Logical negation	.NOT. x_2	True if x_2 is false
.AND.	Logical conjunction	x_1 .AND. x_2	True if x_1 and x_2 are both true
.OR.	Logical inclusive disjunction	x_1 .OR. x_2	True if x_1 and/or x_2 is true
.EQV.	Logical equivalence	x_1 .EQV. x_2	True if both x_1 and x_2 are true or both are false
.NEQV.	Logical nonequivalence	x_1 .NEQV. x_2	True if either x_1 or x_2 is true, but not both

3 The values of the logical intrinsic operations are shown in Table 7.6.

Table 7.6: **The values of operations involving logical intrinsic operators**

x_1	x_2	.NOT. x_2	x_1 .AND. x_2	x_1 .OR. x_2	x_1 .EQV. x_2	x_1 .NEQV. x_2
true	true	false	true	true	true	false
true	false	true	false	true	false	true
false	true	false	false	true	false	true
false	false	true	false	false	true	false

7.1.5.4.2 Evaluation of logical intrinsic operations

1 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 7.23

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)
```

2 Two expressions of type logical are logically equivalent if their values are equal for all possible values of their primaries.

7.1.5.5 Relational intrinsic operations

7.1.5.5.1 Interpretation of relational intrinsic operations

1 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 7.1.5.1.

2 The operators <, <=, >, >=, ==, and /= always have the same interpretations as the operators .LT., .LE., .GT., .GE., .EQ., and .NE., respectively.

NOTE 7.24

As shown in Table 7.2, 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 7.24 (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.

3 Evaluation of a relational intrinsic operation produces a default logical result.

4 The interpretation of the relational intrinsic operators is given in Table 7.7, 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 7.7: **Interpretation of the relational intrinsic operators**

Operator	Representing	Use of operator	Interpretation
.LT.	Less than	x_1 .LT. x_2	x_1 less than x_2
<	Less than	$x_1 < x_2$	x_1 less than x_2
.LE.	Less than or equal to	x_1 .LE. x_2	x_1 less than or equal to x_2
<=	Less than or equal to	$x_1 <= x_2$	x_1 less than or equal to x_2
.GT.	Greater than	x_1 .GT. x_2	x_1 greater than x_2
>	Greater than	$x_1 > x_2$	x_1 greater than x_2
.GE.	Greater than or equal to	x_1 .GE. x_2	x_1 greater than or equal to x_2
>=	Greater than or equal to	$x_1 >= x_2$	x_1 greater than or equal to x_2
.EQ.	Equal to	x_1 .EQ. x_2	x_1 equal to x_2
==	Equal to	$x_1 == x_2$	x_1 equal to x_2
.NE.	Not equal to	x_1 .NE. x_2	x_1 not equal to x_2
/=	Not equal to	x_1 /= x_2	x_1 not equal to x_2

5 A numeric relational intrinsic operation is interpreted as having the logical value true if and only if the values of the operands satisfy the relation specified by the operator.

6 In the numeric relational operation

x_1 *rel-op* x_2

7 if the types or kind type parameters of x_1 and x_2 differ, their values are converted to the type and kind type parameter of the expression $x_1 + x_2$ before evaluation.

8 A character relational intrinsic operation is interpreted as having the logical value true if and only if the values of the operands satisfy the relation specified by the operator.

9 For a character relational intrinsic operation, the operands are compared one character at a time in order, beginning with the first character of each character operand. If the operands are of unequal length, the shorter operand is treated as if it were extended on the right with blanks to the length of the longer operand. If both x_1 and x_2 are of zero length, x_1 is equal to x_2; if every character of x_1 is the same as the character in the corresponding position in x_2, x_1 is equal to x_2. Otherwise, at the first position where the character operands differ, the character operand x_1 is considered to be less than x_2 if the character value of x_1 at this position precedes the value of x_2 in the collating sequence (1.3); x_1 is greater than x_2 if the character value of x_1 at this position follows the value of x_2 in the collating sequence.

NOTE 7.25

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.

7.1.5.5.2 Evaluation of relational intrinsic operations

1 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 violated.

2 Two relational intrinsic operations are relationally equivalent if their logical values are equal for all possible values of their primaries.

7.1.6 Defined operations

7.1.6.1 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* x_2 or *intrinsic-operator* x_2 and that is defined by a function and a generic interface (4.5.5, 12.4.3.5).

2 A function defines the unary operation *op* x_2 if

(1) the function is specified with a FUNCTION (12.6.2.2) or ENTRY (12.6.2.6) statement that specifies one dummy argument d_2,

(2) either

(a) a generic interface (12.4.3.2) provides the function with a *generic-spec* of OPERATOR (*op*), or

(b) there is a generic binding (4.5.5) in the declared type of x_2 with a *generic-spec* of OPERATOR (*op*) and there is a corresponding binding to the function in the dynamic type of x_2,

(3) the type of d_2 is compatible with the dynamic type of x_2,

(4) the type parameters, if any, of d_2 match the corresponding type parameters of x_2, and

(5) either

(a) the rank of x_2 matches that of d_2 or

(b) the function is elemental and there is no other function that defines the operation.

3 If d_2 is an array, the shape of x_2 shall match the shape of d_2.

4 A binary defined operation is an operation that has the form x_1 *defined-binary-op* x_2 or x_1 *intrinsic-operator* x_2 and that is defined by a function and a generic interface.

5 A function defines the binary operation x_1 *op* x_2 if

(1) the function is specified with a FUNCTION (12.6.2.2) or ENTRY (12.6.2.6) statement that specifies two dummy arguments, d_1 and d_2,

(2) either

(a) a generic interface (12.4.3.2) provides the function with a *generic-spec* of OPERATOR (*op*), or

(b) there is a generic binding (4.5.5) in the declared type of x_1 or x_2 with a *generic-spec* of OPERATOR (*op*) and there is a corresponding binding to the function 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 function is elemental, x_1 and x_2 are conformable, and there is no other function that defines the operation.

6 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 7.26

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.

7.1.6.2 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.

7.1.6.3 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.

7.1.7 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 7.27

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 (Z)
```

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 7.28

In the examples in Note 7.27, 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 7.29

This restriction is intended to avoid inadvertant deadlock caused by optimization.

7.1.8 Integrity of parentheses

1 The rules for evaluation specified in subclause 7.1.5 state certain conditions under which a processor may evaluate an expression that is different from the one specified by applying the rules given in 7.1.2 and rules for interpretation specified in subclause 7.1.5. However, any expression in parentheses shall be treated as a data entity.

NOTE 7.30

> 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.

7.1.9 Type, type parameters, and shape of an expression

7.1.9.1 General

1 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 (4.4) or a derived type (4.5).

2 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.

R724 *logical-expr* **is** *expr*

C706 (R724) *logical-expr* shall be of type logical.

R725 *default-char-expr* **is** *expr*

C707 (R725) *default-char-expr* shall be default character.

R726 *int-expr* **is** *expr*

C708 (R726) *int-expr* shall be of type integer.

R727 *numeric-expr* **is** *expr*

C709 (R727) *numeric-expr* shall be of type integer, real, or complex.

7.1.9.2 Type, type parameters, and shape of a primary

1 The type, type parameters, and shape of a primary are determined according to whether the primary is a constant, variable, array constructor, structure constructor, function reference, type parameter inquiry, type parameter name, or parenthesized expression. If a primary is a constant, its type, type parameters, and shape are those of the constant. If it is a structure constructor, it is scalar and its type and type parameters are as described in 4.5.10. If it is an array constructor, its type, type parameters, and shape are as described in 4.8. If it is a variable or function reference, its type, type parameters, and shape are those of the variable (5.2, 5.5) or the function reference (12.5.3), respectively. If the function reference is generic (12.4.3.2, 13.5) then its type, type parameters, and shape are those of the specific function referenced, which is determined by the types, type parameters, and ranks of its actual arguments as specified in 12.5.5.2. If it is a type parameter inquiry or type parameter name, it is a scalar integer with the kind of the type parameter.

2 If a primary is a parenthesized expression, its type, type parameters, and shape are those of the expression.

3 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.

4 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, or as the target in a pointer assignment statement.

5 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 (13.7.126) depend on context.

7.1.9.3 Type, type parameters, and shape of the result of an operation

1 The type of the result of an intrinsic operation $[x_1]$ *op* x_2 is specified by Table 7.2. 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.

2 The type, type parameters, and shape of the result of a defined operation $[x_1]$ *op* x_2 are specified by the function defining the operation (7.1.6).

3 An expression of an intrinsic type has a kind type parameter. An expression of type character also has a character length parameter.

4 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 *op* x_2 where *op* is a numeric intrinsic binary operator with one operand of type integer and the other of type real or complex, the kind type parameter of the expression is that of the real or complex operand.
- For an expression x_1 *op* x_2 where *op* is a numeric intrinsic binary operator with both operands of the same type and kind type parameters, or with one real and one complex with 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 *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 *op* x_2 where *op* is a relational intrinsic operator, the expression has the default logical kind type parameter.

7.1.10 Conformability rules for elemental operations

1 An elemental operation is an intrinsic operation or a defined operation for which the function is elemental (12.8).

2 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.

7.1.11 Specification expression

1 A specification expression is an expression with limitations that make it suitable for use in specifications such as length type parameters (C404) and array bounds (R517, R518). A *specification-expr* shall be a constant expression unless it is in an interface body (12.4.3.2), the specification part of a subprogram or BLOCK construct, a derived type definition, or the *declaration-type-spec* of a FUNCTION statement (12.6.2.2).

R728 *specification-expr* **is** *scalar-int-expr*

C710 (R728) The *scalar-int-expr* shall be a restricted expression.

2 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 object designator with a base object that is a local variable of the procedure containing the BLOCK construct in which the restricted expression appears,
(6) an object designator with a base object that is a local variable of an outer BLOCK construct containing the BLOCK construct in which the restricted expression appears,
(7) an array constructor where each element and each *scalar-int-expr* of each *ac-implied-do-control* is a restricted expression,
(8) a structure constructor where each component is a restricted expression,
(9) 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,
(10) a specification inquiry that is a constant expression,
(11) a reference to the intrinsic function PRESENT,
(12) a reference to any other standard intrinsic function where each argument is a restricted expression,
(13) a reference to a specification function where each argument is a restricted expression,
(14) a type parameter of the derived type being defined,
(15) 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
(16) a restricted expression enclosed in parentheses,

3 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.

4 A specification inquiry is a reference to

(1) an intrinsic inquiry function other than PRESENT,
(2) a type parameter inquiry (6.4.5),
(3) an inquiry function from the intrinsic modules IEEE_ARITHMETIC and IEEE_EXCEPTIONS (14.10),
(4) the function C_SIZEOF from the intrinsic module ISO_C_BINDING (15.2.3.7), or
(5) the COMPILER_VERSION or COMPILER_OPTIONS function from the intrinsic module ISO_FORTRAN_ENV (13.8.2.6, 13.8.2.7).

5 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.

6 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 7.31

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.

7 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.

8 If a specification 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*. 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.

9 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 7.32

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
```

7.1.12 Constant expression

1 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,
(4) 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 the transformational function IEEE_SELECTED_REAL_KIND from the intrinsic module IEEE_ARITHMETIC(14), 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.

R729 *constant-expr* **is** *expr*

C711 (R729) *constant-expr* shall be a constant expression.

R730 *default-char-constant-expr* **is** *default-char-expr*

C712 (R730) *default-char-constant-expr* shall be a constant expression.

R731 *int-constant-expr* **is** *int-expr*

C713 (R731) *int-constant-expr* shall be a constant expression.

2 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*.

3 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 7.33

The following are examples of constant expressions:

```
3
-3 + 4
'AB'
'AB' // 'CD'
('AB' // 'CD') // 'EF'
SIZE (A)
DIGITS (X) + 4
4.0 * atan(1.0)
ceiling(number_of_decimal_digits / log10(radix(0.0)))
```

NOTE 7.33 (cont.)

where A is an explicit-shape array with constant bounds and X is default real.

7.2 Assignment

7.2.1 Assignment statement

7.2.1.1 General form

R732 *assignment-stmt* **is** *variable* = *expr*

C714 (R732) The *variable* shall not be a whole assumed-size array.

NOTE 7.34

Examples of an assignment statement are:

```
A = 3.5 + X * Y
I = INT (A)
```

1 An *assignment-stmt* shall meet the requirements of either a defined assignment statement or an intrinsic assignment statement.

7.2.1.2 Intrinsic assignment statement

1 An intrinsic assignment statement is an assignment statement that is not a defined assignment statement (7.2.1.4). In an intrinsic assignment statement,

(1) if the variable is polymorphic it shall be allocatable and not a coarray,

(2) if *expr* is an array then the variable shall also be an array,

(3) the variable and *expr* shall be conformable unless the variable is an allocatable array that has the same rank as *expr* and is neither a coarray nor a coindexed object,

(4) if the variable is polymorphic it shall be type compatible with *expr*; otherwise the declared types of the variable and *expr* shall conform as specified in Table 7.8,

(5) if the variable is of type character and of ISO 10646, ASCII, or default character kind, *expr* shall be of ISO 10646, ASCII, or default character kind,

(6) otherwise if the variable is of type character *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 *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 *expr* unless the variable is allocatable, is not a coarray, and its corresponding type parameter is deferred.

Table 7.8: **Type conformance for the intrinsic assignment statement**

Type of the variable	Type of *expr*
integer	integer, real, complex
real	integer, real, complex
complex	integer, real, complex
character	character
logical	logical
derived type	same derived type as the variable

2 If *variable* is a coindexed object, the variable

- shall not be polymorphic,
- shall not have an allocatable ultimate component, and
- each deferred length type parameter shall have the same value as the corresponding type parameter of *expr*.

3 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.

7.2.1.3 Interpretation of intrinsic assignments

1 Execution of an intrinsic assignment causes, in effect, the evaluation of the expression *expr* and all expressions within *variable* (7.1), the possible conversion of *expr* to the type and type parameters of the variable (Table 7.9), 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*. No value is assigned to the variable if it is of type character and zero length, or is an array of size zero.

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

- if the variable is polymorphic, the same dynamic type as *expr*,
- each deferred type parameter equal to the corresponding type parameter of *expr*,
- if the variable is an array and *expr* is scalar, the same bounds as before, and
- if *expr* is an array, the shape of *expr* with each lower bound equal to the corresponding element of LBOUND (*expr*).

NOTE 7.35

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, for the assignment statement

```
NAME(:) = 'Dr. '//FIRST_NAME//' '//SURNAME
```

NAME must already be allocated at the time of the assignment; the assigned value is truncated or blank padded to the previously allocated length of NAME.

4 Both *variable* and *expr* may contain references to any portion of the variable.

NOTE 7.36

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

NOTE 7.36 (cont.)

STRING (1:4). If the value of STRING prior to the assignment was 'ABCDEF', the value following the assignment is 'AABCDF'.

5 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*.

6 If the variable is an array, the assignment is performed element-by-element on corresponding array elements of the variable and *expr*.

NOTE 7.37

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.

7 The processor may perform the element-by-element assignment in any order.

NOTE 7.38

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)
```

8 For an intrinsic assignment statement where the variable is of numeric type, the *expr* may 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 7.9.

Table 7.9: **Numeric conversion and the assignment statement**

Type of the variable	Value Assigned
integer	INT (*expr*, KIND = KIND (*variable*))
real	REAL (*expr*, KIND = KIND (*variable*))
complex	CMPLX (*expr*, KIND = KIND (*variable*))
Note: INT, REAL, CMPLX, and KIND are the generic names of functions defined in 13.7.	

9 For an intrinsic assignment statement where the variable is of type logical, the *expr* may have a different kind type parameter, in which case the value of *expr* is converted to the kind type parameter of the variable.

10 For an intrinsic assignment statement where the variable is of type character, the *expr* may 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.

11 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 7.39

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.

12 For an intrinsic assignment of the type C_PTR or C_FUNPTR, the variable becomes undefined if the variable and *expr* are not on the same image.

NOTE 7.40

An intrinsic assignment statement for a variable of type C_PTR or C_FUNPTR is not permitted to involve a coindexed object, see C614, 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, in which case the copy is regarded as undefined.

13 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 (7.2.2) 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.

14 The processor may perform the component-by-component assignment in any order or by any means that has the same effect.

NOTE 7.41

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 7.42

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.

7.2.1.4 Defined assignment statement

1 A defined assignment statement is an assignment statement that is defined by a subroutine and a generic interface (4.5.5, 12.4.3.5.3) that specifies ASSIGNMENT (=).

2 A subroutine defines the defined assignment $x_1 = x_2$ if

(1) the subroutine is specified with a SUBROUTINE (12.6.2.3) or ENTRY (12.6.2.6) statement that specifies two dummy arguments, d_1 and d_2,

(2) either

(a) a generic interface (12.4.3.2) provides the subroutine with a *generic-spec* of ASSIGNMENT (=), or

(b) there is a generic binding (4.5.5) in the declared type of x_1 or x_2 with a *generic-spec* of 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_1 and x_2 are conformable, 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.

7.2.1.5 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 7.43

The rules of defined assignment (12.4.3.5.3), procedure references (12.5), subroutine references (12.5.4), and elemental subroutine arguments (12.8.3) 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.

7.2.2 Pointer assignment

7.2.2.1 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 (7.2.1.3).

7.2.2.2 Syntax of the pointer assignment statement

R733	*pointer-assignment-stmt*	**is**	*data-pointer-object* [(*bounds-spec*-*list*)] => *data-target*
		or	*data-pointer-object* (*bounds-remapping*-*list*) => *data-target*
		or	*proc-pointer-object* => *proc-target*

R734 *data-pointer-object* **is** *variable-name*
or *scalar-variable* % *data-pointer-component-name*

C715 (R733) If *data-target* is not unlimited polymorphic, *data-pointer-object* shall be type compatible (4.3.2.3) with it and the corresponding kind type parameters shall be equal.

C716 (R733) 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.

C717 (R733) If *bounds-spec-list* is specified, the number of *bounds-spec*s shall equal the rank of *data-pointer-object*.

C718 (R733) If *bounds-remapping-list* is specified, the number of *bounds-remapping*s shall equal the rank of *data-pointer-object*.

C719 (R733) If *bounds-remapping-list* is not specified, the ranks of *data-pointer-object* and *data-target* shall be the same.

C720 (R733) A coarray *data-target* shall have the VOLATILE attribute if and only if the *data-pointer-object* has the VOLATILE attribute.

C721 (R734) A *variable-name* shall have the POINTER attribute.

C722 (R734) A *scalar-variable* shall be a *data-ref*.

C723 (R734) A *data-pointer-component-name* shall be the name of a component of *scalar-variable* that is a data pointer.

C724 (R734) A *data-pointer-object* shall not be a coindexed object.

R735 *bounds-spec* **is** *lower-bound-expr* :

R736 *bounds-remapping* **is** *lower-bound-expr* : *upper-bound-expr*

R737 *data-target* **is** *expr*

C725 (R737) 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.

C726 (R737) A *data-target* shall not be a coindexed object.

NOTE 7.44

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, Z[P]%PTR is a reference to the target of component PTR of Z on image P. This target is on image P and its association with Z[P]%PTR must have been established by the execution of an ALLOCATE statement or a pointer assignment on image P.

R738 *proc-pointer-object* **is** *proc-pointer-name*
or *proc-component-ref*

R739 *proc-component-ref* **is** *scalar-variable* % *procedure-component-name*

C727 (R739) The *scalar-variable* shall be a *data-ref* that is not a coindexed object.

C728 (R739) The *procedure-component-name* shall be the name of a procedure pointer component of the declared type of *scalar-variable*.

R740 *proc-target* **is** *expr*

or *procedure-name*
or *proc-component-ref*

C729 (R740) An *expr* shall be a reference to a function whose result is a procedure pointer.

C730 (R740) A *procedure-name* shall be the name of an internal, module, or dummy procedure, a procedure pointer, a specific intrinsic function listed in Table 13.2, 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.

C731 (R740) 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*.

7.2.2.3 Data pointer assignment

1 If the pointer object is not polymorphic (4.3.2.3) 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 7.45

A pointer assignment statement is not permitted to involve a coindexed pointer or target, see C724 and C726. 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 (6.5.4) 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 (13.7.91) 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.

7.2.2.4 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 (12.6.2.4) 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.

7.2.2.5 Examples

NOTE 7.46

The following are examples of pointer assignment statements. (See Note 12.18 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 7.47

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
```

7.2.3 Masked array assignment – WHERE

7.2.3.1 General form of the masked array assignment

1 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.

R741 *where-stmt* **is** WHERE (*mask-expr*) *where-assignment-stmt*

R742 *where-construct* **is** *where-construct-stmt*
    [*where-body-construct*] ...
  [*masked-elsewhere-stmt*
    [*where-body-construct*] ...] ...
  [*elsewhere-stmt*
    [*where-body-construct*] ...]
  end-where-stmt

R743 *where-construct-stmt* **is** [*where-construct-name*:] WHERE (*mask-expr*)

R744 *where-body-construct* **is** *where-assignment-stmt*
  or *where-stmt*
  or *where-construct*

R745 *where-assignment-stmt* **is** *assignment-stmt*

R746 *mask-expr* **is** *logical-expr*

R747 *masked-elsewhere-stmt* **is** ELSEWHERE (*mask-expr*) [*where-construct-name*]

R748 *elsewhere-stmt* **is** ELSEWHERE [*where-construct-name*]

R749 *end-where-stmt* **is** END WHERE [*where-construct-name*]

C732 (R745) A *where-assignment-stmt* that is a defined assignment shall be elemental.

C733 (R742) 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*.

C734 (R744) A statement that is part of a *where-body-construct* shall not be a branch target statement.

2 If a *where-construct* contains a *where-stmt*, a *masked-elsewhere-stmt*, or another *where-construct* then each *mask-expr* within the *where-construct* shall have the same shape. In each *where-assignment-stmt*, the *mask-expr* and the variable being defined shall be arrays of the same shape.

NOTE 7.48

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
```

7.2.3.2 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*).
(3) The control mask m_c is established to have the value m_c .AND. *mask-expr*.

4 The *mask-expr* is evaluated at most once.

5 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.

6 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 7.49

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
```

NOTE 7.49 (cont.)

```
   . . .
   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.

7 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 .AND. (.NOT. *mask-expr*). The control mask is then established to have the value m_c .AND. *mask-expr*. The *mask-expr* is evaluated at most once.

8 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 .AND. *mask-expr*. The pending control mask is not altered.

9 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*.

10 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.

11 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.

12 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.

13 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 7.50

Examples of function references in masked array assignments are:

```
   WHERE (A > 0.0)
      A = LOG (A)            ! LOG is invoked only for positive elements.
      A = A / SUM (LOG (A))  ! LOG is invoked for all elements
                             ! because SUM is transformational
   END WHERE
```

7.2.4 FORALL

7.2.4.1 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*.

R750 *forall-construct* **is** *forall-construct-stmt*

[*forall-body-construct*] ...
end-forall-stmt

R751 *forall-construct-stmt* **is** [*forall-construct-name* :] FORALL *concurrent-header*

R752 *forall-body-construct* **is** *forall-assignment-stmt*
or *where-stmt*
or *where-construct*
or *forall-construct*
or *forall-stmt*

R753 *forall-assignment-stmt* **is** *assignment-stmt*
or *pointer-assignment-stmt*

R754 *end-forall-stmt* **is** END FORALL [*forall-construct-name*]

C735 (R754) 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*.

C736 (R752) A statement in a *forall-body-construct* shall not define an *index-name* of the *forall-construct*.

C737 (R752) Any procedure referenced in a *forall-body-construct*, including one referenced by a defined operation, assignment, or finalization, shall be a pure procedure.

C738 (R752) 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 16.4.

7.2.4.2 Execution of the FORALL construct

7.2.4.2.1 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.

7.2.4.2.2 Determination of the values for index variables

1 The values of the index variables are determined as they are for the DO CONCURRENT statement (8.1.6.4.2).

7.2.4.2.3 Evaluation of the mask expression

1 The mask expression is evaluated as it is for the DO CONCURRENT statement (8.1.6.4.2).

7.2.4.2.4 Execution of the FORALL body constructs

1 The *forall-body-construct*s 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:

2 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.

3 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.

4 In a *forall-assignment-stmt*, a defined assignment subroutine shall not reference any *variable* that becomes defined by the statement.

NOTE 7.51

If a variable defined in an assignment statement within a FORALL construct is referenced in a later statement in that construct, the later statement uses the value(s) computed in the preceding assignment statement, not the value(s) the variable had prior to execution of the FORALL.

5 Each statement in a *where-construct* (7.2.3) 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-construct*s, masked by any control mask corresponding to outer *where-construct*s. 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*.

6 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.

7.2.4.3 The FORALL statement

1 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.

R755 *forall-stmt* **is** FORALL *concurrent-header forall-assignment-stmt*

2 A FORALL statement is equivalent to a FORALL construct containing a single *forall-body-construct* that is a *forall-assignment-stmt*.

3 The scope of an *index-name* in a *forall-stmt* is the statement itself (16.4).

7.2.4.4 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 7.52

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

```
FORALL (I = 1:10)
   A (INDEX (I)) = B(I)
END FORALL
```

if and only if INDEX(1:10) contains no repeated values.

2 Within the scope of a FORALL construct, a nested FORALL statement or FORALL construct shall not have the same *index-name*. The *concurrent-header* expressions within a nested FORALL may depend on the values of outer *index-name* variables.

(Blank page)

8 Execution control

8.1 Executable constructs containing blocks

8.1.1 Blocks

1 The following are executable constructs that contain blocks:

- ASSOCIATE construct;
- BLOCK construct;
- CRITICAL construct;
- DO construct;
- IF construct;
- SELECT CASE construct;
- SELECT TYPE construct.

R801 *block* **is** [*execution-part-construct*] ...

2 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 8.1

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
```

8.1.2 Rules governing blocks

8.1.2.1 Control flow in blocks

1 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.

2 Subroutine and function references (12.5.3, 12.5.4) may appear in a block.

8.1.2.2 Execution of a block

1 Execution of a block begins with the execution of the first executable construct in the block.

2 Execution of the block is completed when

- the last executable construct in the sequence is executed,
- a branch (8.2) within the block that has a branch target outside the block occurs,
- a RETURN statement within the block is executed,
- an EXIT statement within the block that belongs to the block or an outer construct is executed, or
- a CYCLE statement within the block that belongs to an outer construct is executed.

NOTE 8.2

The action that takes place at the terminal boundary depends on the particular construct and on the block within that construct.

8.1.3 ASSOCIATE construct

8.1.3.1 Purpose and form of the ASSOCIATE construct

1 The ASSOCIATE construct associates named entities with expressions or variables during the execution of its block. These named construct entities (16.4) are associating entities (16.5.1.6). The names are associate names.

R802 *associate-construct* **is** *associate-stmt*
block
end-associate-stmt

R803 *associate-stmt* **is** [*associate-construct-name* :] ASSOCIATE ■
■ (*association-list*)

R804 *association* **is** *associate-name* => *selector*

R805 *selector* **is** *expr*
or *variable*

C801 (R804) If *selector* is not a *variable* or is a *variable* that has a vector subscript, *associate-name* shall not appear in a variable definition context (16.6.7).

C802 (R804) An *associate-name* shall not be the same as another *associate-name* in the same *associate-stmt*.

C803 (R805) *variable* shall not be a coindexed object.

C804 (R805) *expr* shall not be a variable.

C805 (R805) *expr* shall not be a *designator* of a procedure pointer or a function reference that returns a procedure pointer.

R806 *end-associate-stmt* **is** END ASSOCIATE [*associate-construct-name*]

C806 (R806) If the *associate-stmt* of an *associate-construct* specifies an *associate-construct-name*, the corresponding *end-associate-stmt* shall specify the same *associate-construct-name*. If the *associate-stmt* of an *associate-construct* does not specify an *associate-construct-name*, the corresponding *end-associate-stmt* shall not specify an *associate-construct-name*.

8.1.3.2 Execution of the ASSOCIATE construct

1 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 (16.5.1.6) 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.

2 The other attributes of the associating entity are described in 8.1.3.3.

3 It is permissible to branch to an *end-associate-stmt* only from within its ASSOCIATE construct.

8.1.3.3 Other attributes of associate names

1 Within an ASSOCIATE or SELECT TYPE construct, each associating entity has the same rank and corank as its associated selector. The lower bound of each dimension is the result of the intrinsic function LBOUND (13.7.91) 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 cobounds of each codimension of the associating entity are the same as those of the selector. The associating entity has the ASYNCHRONOUS or VOLATILE attribute if and only if the selector is a variable and has the attribute. The associating entity has the TARGET attribute if and only if the selector is a variable and has either the TARGET or POINTER 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.

2 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 defined or become undefined. If the selector is not permitted to appear in a variable definition context (16.6.7), the associate name shall not appear in a variable definition context.

8.1.3.4 Examples of the ASSOCIATE construct

NOTE 8.3

The following example illustrates an association with an expression.

```
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.

```
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.

```
ASSOCIATE ( ARRAY => AX%B(I,:)%C )
  ARRAY(N)%EV = ARRAY(N-1)%EV
END ASSOCIATE
```

The following example illustrates multiple associations.

```
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
```

8.1.4 BLOCK construct

1 The BLOCK construct is an executable construct that may contain declarations.

R807 *block-construct* **is** *block-stmt*
  [*specification-part*]
  block
  end-block-stmt

R808 *block-stmt* **is** [*block-construct-name* :] BLOCK

R809 *end-block-stmt* **is** END BLOCK [*block-construct-name*]

C807 (R807) The *specification-part* of a BLOCK construct shall not contain a COMMON, EQUIVALENCE, IMPLICIT, INTENT, NAMELIST, OPTIONAL, statement function, or VALUE statement.

C808 (R807) A SAVE statement in a BLOCK construct shall contain a *saved-entity-list* that does not specify a *common-block-name*.

C809 (R807) 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 (16.4). 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.

NOTE 8.4

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, because a SAVE statement affects variables in its scoping unit rather than in its inclusive scope. For example,

```
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
```

8.1.5 CRITICAL construct

1 A CRITICAL construct limits execution of a block to one image at a time.

R810 *critical-construct* **is** *critical-stmt*
block

end-critical-stmt

R811 *critical-stmt* **is** [*critical-construct-name* :] CRITICAL

R812 *end-critical-stmt* **is** END CRITICAL [*critical-construct-name*]

C810 (R810) 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*.

C811 (R810) The *block* of a *critical-construct* shall not contain a RETURN statement or an image control statement.

C812 A branch (8.2) 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. 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 executing *block*. The image shall not execute an image control statement during the execution of *block*. The sequence of executed statements is therefore a segment (8.5.2). If image T is the next to execute the construct after image M, the segment on image M precedes the segment on image T.

NOTE 8.5

If more than one image executes the block of a CRITICAL construct, its execution by one image always either precedes or succeeds its execution by another image. Typically no other statement ordering is needed. Consider the following example:

```
CRITICAL
    GLOBAL_COUNTER[1] = GLOBAL_COUNTER[1] + 1
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 8.6

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
DO
   CRITICAL
      JOB = NUM_JOBS[1]
      NUM_JOBS[1] = JOB - 1
   END CRITICAL
   IF (JOB > 0) THEN
      ! Work on JOB
   ELSE
      EXIT
   END IF
END DO
SYNC ALL
```

8.1.6 DO construct

8.1.6.1 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 (8.1.6.4.5). An iteration of the loop can be curtailed by executing a CYCLE statement (8.1.6.4.4).

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 16.4.

8.1.6.2 Form of the DO construct

R813	*do-construct*	**is**	*do-stmt*
			    block
			    end-do

R814	*do-stmt*	**is**	*nonlabel-do-stmt*
		or	*label-do-stmt*
R815	*label-do-stmt*	**is**	[*do-construct-name* :] DO *label* [*loop-control*]
R816	*nonlabel-do-stmt*	**is**	[*do-construct-name* :] DO [*loop-control*]
R817	*loop-control*	**is**	[,] *do-variable* = *scalar-int-expr*, *scalar-int-expr* ■ ■ [, *scalar-int-expr*]
		or	[,] WHILE (*scalar-logical-expr*)
		or	[,] CONCURRENT *concurrent-header*
R818	*do-variable*	**is**	*scalar-int-variable-name*

C813 (R818) The *do-variable* shall be a variable of type integer.

R819	*concurrent-header*	**is**	([*integer-type-spec* ::] *concurrent-control-list* [, *scalar-mask-expr*])
R820	*concurrent-control*	**is**	*index-name* = *concurrent-limit* : *concurrent-limit* [: *concurrent-step*]
R821	*concurrent-limit*	**is**	*scalar-int-expr*
R822	*concurrent-step*	**is**	*scalar-int-expr*

C814 (R819) Any procedure referenced in the *scalar-mask-expr*, including one referenced by a defined operation, shall be a pure procedure (12.7).

C815 (R820) The *index-name* shall be a named scalar variable of type integer.

C816 (R820) 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.

R823	*end-do*	**is**	*end-do-stmt*
		or	*continue-stmt*
R824	*end-do-stmt*	**is**	END DO [*do-construct-name*]

C817 (R813) 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*.

C818 (R813) If the *do-stmt* is a *nonlabel-do-stmt*, the corresponding *end-do* shall be an *end-do-stmt*.

C819 (R813) If the *do-stmt* is a *label-do-stmt*, the corresponding *end-do* shall be identified with the same *label*.

1 It is permissible to branch to an *end-do* only from within its DO construct.

8.1.6.3 Active and inactive DO constructs

1 A DO construct is either active or inactive. Initially inactive, a DO construct becomes active only when its DO statement is executed.

2 Once active, the DO construct becomes inactive only when it terminates (8.1.6.4.5).

8.1.6.4 Execution of a DO construct

8.1.6.4.1 Loop initiation

1 When the DO statement is executed, the DO construct becomes active. If *loop-control* is

[,] *do-variable* = *scalar-int-expr*$_1$, *scalar-int-expr*$_2$ [, *scalar-int-expr*$_3$]

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-expr*$_1$, *scalar-int-expr*$_2$, and *scalar-int-expr*$_3$, respectively, including, if necessary, conversion to the kind type parameter of the *do-variable* according to the rules for numeric conversion (Table 7.9). If *scalar-int-expr*$_3$ 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 8.7

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 8.1.6.4.2.

4 At the completion of the execution of the DO statement, the execution cycle begins.

8.1.6.4.2 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 7.9). If *concurrent-step* does not appear, m_3 has the value 1. The value m_3 shall not be zero.

(2) Let the value of *max* be $(m_2 - m_1 + m_3)/m_3$. If *max*≤ 0 for some *index-name*, the execution of the construct is complete. Otherwise, the set of values for the *index-name* is

$$m_1 + (k - 1) \times m_3 \qquad \text{where } k = 1, 2, ..., max.$$

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.

3 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*.

4 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 8.8

The *index-name* variables can appear in the mask, for example

```
DO CONCURRENT (I=1:10, J=1:10, A(I) > 0.0 .AND. B(J) < 1.0)
. . .
```

8.1.6.4.3 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) The iteration count, if any, is tested. If it is zero, the loop terminates and the DO construct becomes inactive. If *loop-control* is [,] WHILE (*scalar-logical-expr*), the *scalar-logical-expr* is evaluated; if the value of this expression is false, the loop terminates and the DO construct becomes inactive.

(2) The *block* of the loop is executed.

(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_3.

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.

8.1.6.4.4 CYCLE statement

1 Execution of a loop iteration can be curtailed by executing a CYCLE statement from that belongs to the construct.

R825 *cycle-stmt* **is** CYCLE [*do-construct-name*]

C820 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*.

C821 A *cycle-stmt* shall not appear within a 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.

8.1.6.4.5 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.

8.1.6.5 Restrictions on DO CONCURRENT constructs

C822 A RETURN statement shall not appear within a DO CONCURRENT construct.

C823 An image control statement shall not appear within a DO CONCURRENT construct.

C824 A branch (8.2) within a DO CONCURRENT construct shall not have a branch target that is outside the construct.

C825 A reference to a nonpure procedure shall not appear within a DO CONCURRENT construct.

C826 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 following additional restrictions apply to execution of a DO CONCURRENT construct.

- A variable that is referenced in an iteration shall either be previously defined during that iteration, or shall not be defined or become undefined during any other iteration. A variable that is defined or becomes undefined by more than one iteration becomes undefined when the loop terminates.
- A pointer that is used in an iteration other than as the pointer in pointer assignment, allocation, or nullification, shall either be previously pointer associated during that iteration or shall not have its pointer association changed during any iteration. A pointer that has its pointer association changed in more than one iteration has an association status of undefined when the construct terminates.
- If an allocatable object is allocated in more than one iteration, it shall have an allocation status of unallocated at the end of every iteration. An allocatable object that is referenced, defined, deallocated, or has its allocation status, dynamic type, or a deferred type parameter value inquired about, in any iteration, shall either be previously allocated in that iteration or shall not be allocated or deallocated in any other iteration.

- 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.

2 If records are written to a file connected for sequential access by more than one iteration, the ordering between records written by different iterations is processor dependent.

NOTE 8.9

The restrictions on referencing variables defined in an iteration of a DO CONCURRENT construct apply to any procedure invoked within the loop.

NOTE 8.10

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.

NOTE 8.11

A variable that is effectively local to each iteration of a DO CONCURRENT construct can be declared in a BLOCK construct within it. For example:

```
DO CONCURRENT (I = 1:N)
   BLOCK
      REAL :: T
      T = A(I) + B(I)
      C(I) = T + SQRT(T)
   END BLOCK
END DO
```

8.1.6.6 Examples of DO constructs

NOTE 8.12

The following program fragment computes a tensor product of two arrays:

```
DO I = 1, M
   DO J = 1, N
      C (I, J) = DOT_PRODUCT (A (I, J, :), B(:, I, J))
   END DO
END DO
```

NOTE 8.13

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.

```
READ (IUN, '(1X, G14.7)', IOSTAT = IOS) X
DO WHILE (IOS == 0)
   IF (X >= 0.) THEN
      CALL SUBA (X)
      CALL SUBB (X)
          ...
      CALL SUBZ (X)
   ENDIF
   READ (IUN, '(1X, G14.7)', IOSTAT = IOS) X
END DO
```

NOTE 8.14

The following example behaves exactly the same as the one in Note 8.13. 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 8.15

The following example represents 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 8.16

Additional examples of DO constructs are in C.5.3.

8.1.7 IF construct and statement

8.1.7.1 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.

R826 *if-construct* **is** *if-then-stmt*
      block
    [*else-if-stmt*
      block] ...
    [*else-stmt*
      block]
    end-if-stmt

R827 *if-then-stmt* **is** [*if-construct-name* :] IF (*scalar-logical-expr*) THEN

R828 *else-if-stmt* **is** ELSE IF (*scalar-logical-expr*) THEN [*if-construct-name*]

R829 *else-stmt* **is** ELSE [*if-construct-name*]

R830 *end-if-stmt* **is** END IF [*if-construct-name*]

C827 (R826) 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*.

8.1.7.2 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.

8.1.7.3 Examples of IF constructs

NOTE 8.17

```
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
   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
```

8.1.7.4 IF statement

1 The IF statement controls the execution of a single action statement based on a single logical expression.

R831 *if-stmt* **is** IF (*scalar-logical-expr*) *action-stmt*

C828 (R831) The *action-stmt* in the *if-stmt* shall not be an *end-function-stmt*, *end-mp-subprogram-stmt*, *end-program-stmt*, *end-subroutine-stmt*, or *if-stmt*.

2 Execution of an IF statement causes evaluation of the scalar logical expression. If the value of the expression is true, the action statement is executed. If the value is false, the action statement is not executed and execution continues.

3 The execution of a function reference in the scalar logical expression may affect entities in the action statement.

NOTE 8.18

An example of an IF statement is:

```
IF (A > 0.0) A = LOG (A)
```

8.1.8 SELECT CASE construct

8.1.8.1 Purpose and form of the SELECT CASE construct

1 The SELECT CASE construct selects for execution at most one of its constituent blocks. The selection is based on the value of an expression.

R832 *case-construct* **is** *select-case-stmt*
  [*case-stmt*
    block] ...
  end-select-stmt

R833 *select-case-stmt* **is** [*case-construct-name* :] SELECT CASE (*case-expr*)

R834 *case-stmt* **is** CASE *case-selector* [*case-construct-name*]

R835 *end-select-stmt* **is** END SELECT [*case-construct-name*]

C829 (R832) If the *select-case-stmt* of a *case-construct* specifies a *case-construct-name*, the corresponding *end-select-stmt* shall specify the same *case-construct-name*. If the *select-case-stmt* of a *case-construct* does not specify a *case-construct-name*, the corresponding *end-select-stmt* shall not specify a *case-construct-name*. If a *case-stmt* specifies a *case-construct-name*, the corresponding *select-case-stmt* shall specify the same *case-construct-name*.

R836 *case-expr* **is** *scalar-expr*

C830 *case-expr* shall be of type character, integer, or logical.

R837 *case-selector* **is** (*case-value-range-list*)
  or DEFAULT

C831 (R832) No more than one of the selectors of one of the CASE statements shall be DEFAULT.

R838 *case-value-range* **is** *case-value*
  or *case-value* :
  or : *case-value*
  or *case-value* : *case-value*

R839 *case-value* **is** *scalar-constant-expr*

C832 (R832) 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.

C833 (R832) A *case-value-range* using a colon shall not be used if *case-expr* is of type logical.

C834 (R832) For a given *case-construct*, there shall be no possible value of the *case-expr* that matches more than one *case-value-range*.

8.1.8.2 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.

8.1.8.3 Examples of SELECT CASE constructs

NOTE 8.19

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 8.20

A code fragment to check for balanced parentheses:

```
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 8.21

The following three fragments are equivalent:

```
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 THAT
CASE (1)
   CALL THIS
END SELECT
```

NOTE 8.22

A code fragment showing several selections of one block:

```
SELECT CASE (N)
CASE (1, 3:5, 8)  ! Selects 1, 3, 4, 5, 8
   CALL SUB
CASE DEFAULT
   CALL OTHER
END SELECT
```

8.1.9 SELECT TYPE construct

8.1.9.1 Purpose and form of the SELECT TYPE construct

1 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 (16.4, 16.5.1.6), in the same way as for the ASSOCIATE construct.

R840 *select-type-construct* **is** *select-type-stmt*
  [*type-guard-stmt*
    block] ...
  end-select-type-stmt

R841 *select-type-stmt* **is** [*select-construct-name* :] SELECT TYPE ■
  ■ ([*associate-name* =>] *selector*)

C835 (R841) If *selector* is not a named *variable*, *associate-name* => shall appear.

C836 (R841) If *selector* is not a *variable* or is a *variable* that has a vector subscript, *associate-name* shall not appear in a variable definition context (16.6.7).

C837 (R841) The *selector* in a *select-type-stmt* shall be polymorphic.

R842 *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*]

C838 (R842) The *type-spec* or *derived-type-spec* shall specify that each length type parameter is assumed.

C839 (R842) The *type-spec* or *derived-type-spec* shall not specify a type with the BIND attribute or the SEQUENCE attribute.

C840 (R840) 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*.

C841 (R840) 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*.

C842 (R840) For a given *select-type-construct*, there shall be at most one CLASS DEFAULT *type-guard-stmt*.

R843 *end-select-type-stmt* **is** END SELECT [*select-construct-name*]

C843 (R840) 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*.

2 The associate name of a SELECT TYPE construct is the *associate-name* if specified; otherwise it is the *name* that constitutes the *selector*.

8.1.9.2 Execution of the SELECT TYPE construct

1 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.

2 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 (16.5.1.6) with the selector.

3 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.

4 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 must 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 8.23

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.

5 Within the block following a TYPE IS type guard statement, the associating entity (16.5.5) is not polymorphic (4.3.2.3), has the type named in the type guard statement, and has the type parameter values of the selector.

6 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.

7 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 8.24

If the declared type of the *selector* is T, specifying CLASS DEFAULT has the same effect as specifying CLASS IS (T).

8 The other attributes of the associating entity are described in 8.1.3.3.

9 It is permissible to branch to an *end-select-type-stmt* only from within its SELECT TYPE construct.

8.1.9.3 Examples of the SELECT TYPE construct

NOTE 8.25

```
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 8.26

The following example illustrates the omission of *associate-name*. It uses the declarations from Note 8.25.

```
P_OR_C => P3
SELECT TYPE ( P_OR_C )
CLASS IS ( POINT )
  ! "CLASS ( POINT ) :: P_OR_C" implied here
```

NOTE 8.26 (cont.)

```
  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
```

8.1.10 EXIT statement

1 The EXIT statement provides one way of terminating a loop, or completing execution of another construct.

R844 *exit-stmt* **is** EXIT [*construct-name*]

C844 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.

C845 An *exit-stmt* shall not appear within a 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 (8.1.6.4.5) 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.

8.2 Branching

8.2.1 Branch concepts

1 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-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*, or *where-construct-stmt*.

8.2.2 GO TO statement

R845 *goto-stmt* **is** GO TO *label*

C846 (R845) The *label* shall be the statement label of a branch target statement that appears in the same inclusive scope as the *goto-stmt*.

1 Execution of a GO TO statement causes a branch to the branch target statement identified by the label.

8.2.3 Computed GO TO statement

R846 *computed-goto-stmt* **is** GO TO (*label-list*) [,] *scalar-int-expr*

C847 (R846) Each *label* in *label-list* shall be the statement label of a branch target statement that appears in the same inclusive scope as the *computed-goto-stmt*.

1 Execution of a computed GO TO statement causes evaluation of the scalar integer expression. If this value is i such that $1 \leq i \leq n$ where n is the number of labels in *label-list*, a branch occurs to the branch target statement identified by the i^{th} label in the list of labels. If i is less than 1 or greater than n, the execution sequence continues as though a CONTINUE statement were executed.

8.3 CONTINUE statement

1 Execution of a CONTINUE statement has no effect.

R847 *continue-stmt* **is** CONTINUE

8.4 STOP and ERROR STOP statements

R848 *stop-stmt* **is** STOP [*stop-code*]

R849 *error-stop-stmt* **is** ERROR STOP [*stop-code*]

R850 *stop-code* **is** *scalar-default-char-constant-expr*
or *scalar-int-constant-expr*

C848 (R850) The *scalar-int-constant-expr* shall be of default kind.

1 Execution of a STOP statement initiates normal termination of execution. Execution of an ERROR STOP statement initiates error termination of execution.

2 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 any exception (14) is signaling on that image, the processor shall issue a warning indicating which exceptions are signaling; this warning shall be on the unit identified by the named constant ERROR_UNIT (13.8.2.8). It is recommended that the stop code is made available by formatted output to the same unit.

NOTE 8.27

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 8.28

If the *stop-code* is an integer, it is recommended that the value also be used as the process exit status, if the processor supports that concept. 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).

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.

8.5 Image execution control

8.5.1 Image control statements

1 The execution sequence on each image is specified in 2.3.5.

2 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;
- ALLOCATE or DEALLOCATE statement that has a coarray *allocate-object*;
- CRITICAL or END CRITICAL (8.1.5);
- 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.

3 All image control statements except CRITICAL, END CRITICAL, LOCK, and UNLOCK include the effect of executing a SYNC MEMORY statement (8.5.5).

4 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.

8.5.2 Segments

1 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.

2 By execution of image control statements or user-defined ordering (8.5.5), the program can ensure that the execution of the i^{th} segment on image P, P_i, either precedes or succeeds the execution of the j^{th} segment on another image Q, 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.

3 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. Otherwise,

- if a variable is defined 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 or defined 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}, ..., 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 8.29

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 on the same image or because of the execution of image control statements.

NOTE 8.30

> If the segments S_1, S_2, ..., S_k on the distinct images P_1, P_2, ..., 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 8.31

> 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 8.32

> 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 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 8.33

> 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.

8.5.3 SYNC ALL statement

R851	*sync-all-stmt*	**is**	SYNC ALL [([*sync-stat-list*])]
R852	*sync-stat*	**is**	STAT = *stat-variable*
		or	ERRMSG = *errmsg-variable*

C849 No specifier shall appear more than once in a given *sync-stat-list*.

1 The STAT= and ERRMSG= specifiers for image control statements are described in 8.5.7.

2 Execution of a SYNC ALL statement performs a synchronization of all images. Execution on an image, M, of the segment following the SYNC ALL statement is delayed until each other image has executed a SYNC ALL statement as many times as has image M. 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 8.34

> 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:

```
REAL :: P[*]
```

NOTE 8.34 (cont.)

```
  ...
SYNC ALL
IF (THIS_IMAGE()==1) THEN
   READ (*,*) P
   DO I = 2, NUM_IMAGES()
      P[I] = P
   END DO
END IF
SYNC ALL
```

8.5.4 SYNC IMAGES statement

R853	*sync-images-stmt*	**is**	SYNC IMAGES (*image-set* [, *sync-stat-list*])
R854	*image-set*	**is**	*int-expr*
		or	*

C850 An *image-set* that is an *int-expr* shall be scalar or of rank one.

1 If *image-set* is an array expression, the value of each element shall be positive and not greater than the number of images, and there shall be no repeated values.

2 If *image-set* is a scalar expression, its value shall be positive and not greater than the number of images.

3 An *image-set* that is an asterisk specifies all images.

4 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 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. 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 8.35

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 8.36

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
```

NOTE 8.36 (cont.)

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>NUM_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 8.37

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.

8.5.5 SYNC MEMORY statement

1 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.

R855 *sync-memory-stmt* **is** SYNC MEMORY [([*sync-stat-list*])]

2 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.

3 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.

4 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 8.38

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 8.39

SYNC MEMORY can be used to implement specialized schemes for segment ordering, such as the spin-wait loop. For example:

```
USE,INTRINSIC :: ISO_FORTRAN_ENV
LOGICAL(ATOMIC_LOGICAL_KIND),SAVE :: LOCKED[*] = .TRUE.
LOGICAL :: VAL
INTEGER :: IAM, P, Q
...
IAM = THIS_IMAGE()
IF (IAM == P) THEN
                                              ! Segment P_i
   SYNC MEMORY                                ! A
   CALL ATOMIC_DEFINE (LOCKED[Q], .FALSE.)    ! Segment P_{i+1}
ELSE IF (IAM == Q) THEN
  VAL = .TRUE.
  DO WHILE (VAL)                              ! Segment Q_{j-1}
     CALL ATOMIC_REF (VAL, LOCKED)
  END DO
  SYNC MEMORY                                 ! B
                                              ! Segment Q_j
END IF
```

The DO WHILE loop does not complete until VAL is defined with the value false. This is the cooperative synchronization that provides the dependency that image Q does not complete segment Q_{j-1} until the CALL statement in segment P_{i+1} completes. This ensures that the execution of segment P_i on image P precedes execution of segment Q_j on image Q.

The first SYNC MEMORY statement (A) ensures that the compiler does not reorder the following statement (segment P_{i+1}) with the previous statements, since the lock is supposed to be freed only after the work in segment P_i has been completed.

The second SYNC MEMORY statement (B) marks the beginning of a new segment, informing the compiler that the values of coarrays referenced in that segment might have been changed by other images in preceding segments, so need to be loaded from memory.

NOTE 8.40

As a second 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:

```
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]
```

where executing the subroutine EXTERNAL_SYNC has an image synchronization effect similar to executing a SYNC ALL statement.

8.5.6 LOCK and UNLOCK statements

R856 *lock-stmt* **is** LOCK (*lock-variable* [, *lock-stat-list*])

R857 *lock-stat* **is** ACQUIRED_LOCK = *scalar-logical-variable*
or *sync-stat*

C851 No specifier shall appear more than once in a given *lock-stat-list*.

R858 *unlock-stmt* **is** UNLOCK (*lock-variable* [, *sync-stat-list*])

R859 *lock-variable* **is** *scalar-variable*

C852 (R859) A *lock-variable* shall be of type LOCK_TYPE (13.8.2.16).

1 A lock variable is unlocked if its value is equal to that of LOCK_TYPE ( ). 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 and has not been subsequently unlocked by execution of an UNLOCK statement on the same image.

2 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 defined after the other image causes the lock variable to become unlocked.

3 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.

4 Successful execution of an UNLOCK statement causes the lock variable to become unlocked.

5 During the 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. 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.

6 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 8.41

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 8.42

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 8.43

The following example illustrates the use of LOCK and UNLOCK statements to manage a work queue:

```
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
   ... ! 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
      work_queue(work_queue_size[me+1])[me+1] = job
   END IF
   UNLOCK (queue_lock[me+1]) ! Ends segment B

END DO
```

8.5.7 STAT= and ERRMSG= specifiers in image control statements

1 If the STAT= specifier appears, successful execution of the LOCK, SYNC ALL, SYNC IMAGES, SYNC MEMORY, or UNLOCK statement causes the specified variable to become defined with the value zero.

2 If the STAT= specifier appears in a SYNC ALL or SYNC IMAGES statement and execution of one of these statements involves synchronization with an image that has initiated termination, the variable becomes defined with the value of the constant STAT_STOPPED_IMAGE (13.8.2.24) in the intrinsic module ISO_FORTRAN_ENV(13.8.2), and the effect of executing the statement is otherwise the same as that of executing the SYNC MEMORY statement. If any other error condition occurs during execution of one of these statements, the variable becomes defined with a processor-dependent positive integer value that is different from the value of STAT_STOPPED_IMAGE.

3 If the STAT= specifier appears in a LOCK statement and the lock variable is locked by the executing image, the specified variable becomes defined with the value of STAT_LOCKED (13.8.2.22). If the STAT= specifier appears in an UNLOCK statement and the lock variable has the value unlocked, the variable specified by the STAT= specifier becomes defined with the value of STAT_UNLOCKED (13.8.2.25). If the STAT= specifier appears in an UNLOCK statement and the lock variable is locked by a different image, the specified variable becomes defined with the value STAT_LOCKED_OTHER_IMAGE (13.8.2.23). The named constants STAT_LOCKED, STAT_UNLOCKED, and STAT_LOCKED_OTHER_IMAGE are defined in 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_UNLOCKED, and STAT_LOCKED_OTHER_IMAGE.

4 If an error condition occurs during execution of a LOCK, SYNC ALL, SYNC IMAGES, SYNC MEMORY, or UNLOCK statement that does not contain the STAT= specifier, error termination is initiated.

5 If an ERRMSG= specifier appears in a LOCK, SYNC ALL, SYNC IMAGES, SYNC MEMORY, or UNLOCK statement, and an error condition occurs during execution of that statement, the specified variable is assigned, as if by intrinsic assignment, an explanatory message. If no such condition occurs, the definition status and value of the specified variable are unchanged.

6 The set of error conditions that can occur in an image control statement is processor dependent.

NOTE 8.44

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.

(Blank page)

9 Input/output statements

9.1 Input/output concepts

1 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.

2 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.

3 The input/output statements are the BACKSPACE, CLOSE, ENDFILE, FLUSH, INQUIRE, OPEN, PRINT, READ, REWIND, WAIT, and WRITE statements.

4 A file is composed of either a sequence of file storage units (9.3.5) 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.

5 A file is either an external file (9.3) or an internal file (9.4).

9.2 Records

9.2.1 Definition of a record

1 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 9.1

What is called a "record" in Fortran is commonly called a "logical record". There is no concept in Fortran of a "physical record."

9.2.2 Formatted record

1 A formatted record consists of a sequence of characters that are representable in the processor; however, a processor may prohibit some control characters (3.1.1) 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.

9.2.3 Unformatted record

1 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

(9.3.5) and depends on the output list (9.6.3) 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.

9.2.4 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 9.2

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 (9.8.2).

9.3 External files

9.3.1 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 9.3

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 9.4

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 (9.3.4).

NOTE 9.5

For more explanatory information on external files, see C.6.1.

9.3.2 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.

2 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.

3 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.

9.3.3 File access

9.3.3.1 File access methods

1 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 9.6

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.

2 The method of accessing a file is determined when the file is connected to a unit (9.5.4) or when the file is created if the file is preconnected (9.5.5).

9.3.3.2 Sequential access

1 Sequential access is a method of accessing the records of an external record file in order.

2 When 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.

9.3.3.3 Direct access

1 Direct access is a method of accessing the records of an external record file in arbitrary order.

2 When 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 can never 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 (10.10), namelist formatting (10.11), or a nonadvancing data transfer statement (9.3.4.2).

NOTE 9.7

A record cannot be deleted; however, a record can be rewritten.

9.3.3.4 Stream access

1 Stream access is a method of accessing the file storage units (9.3.5) of an external stream file.

2 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.

3 When 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.

4 When 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 (9.5.6.15) 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 (3.1.1) from appearing in a formatted stream file.

NOTE 9.8

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 9.9

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 (9.10.2.22).

9.3.4 File position

9.3.4.1 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 ith record or between the $(i-1)$th record and the ith 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 nth 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 ith record or between the ith 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 nth (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.

9.3.4.2 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 (10.8.2). 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. It is also possible to read variable-length records and be notified of their lengths. 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.

9.3.4.3 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.

7 File positioning for child data transfer statements is described in 9.6.4.8.

9.3.4.4 File position after data transfer

1 If an error condition (9.11) occurred, the position of the file is indeterminate. If no error condition occurred, but an end-of-file condition (9.11) 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 9.10

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 (9.11) 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 9.11

The highest-numbered position might not be the current one if the output involved a T, TL, TR, or X edit descriptor (10.8.1) and the statement is a nonadvancing output statement.

9.3.5 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 (9.10.3).

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 (13.8.2.9) 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 9.12

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

NOTE 9.12 (cont.)

never required. A file storage unit of one bit would always meet this goal, but would likely be impractical because of the alignment requirements.

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.

9.4 Internal files

1 Internal files provide a means of transferring and converting data from internal storage to internal storage.

2 An internal file is a record file with the following properties.

- The file is a variable of default, ASCII, or ISO 10646 character 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 (6.5.3.2) or the array section (6.5.3.3). Every record of the file has the same length, which is the length of an array element in the array.
- 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 (9.6.4.8). This record becomes the current record.
- The initial value of a connection mode (9.5.2) 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.

9.5 File connection

9.5.1 Referring to a file

1 A unit, specified by an *io-unit*, provides a means for referring to a file.

R901	*io-unit*	**is**	*file-unit-number*
		or	*
		or	*internal-file-variable*
R902	*file-unit-number*	**is**	*scalar-int-expr*
R903	*internal-file-variable*	**is**	*char-variable*

C901 (R903) The *char-variable* shall not be an array section with a vector subscript.

C902 (R903) The *char-variable* shall be default character, ASCII character, or ISO 10646 character.

2 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 (13.8.2), the `unit` argument of an active defined input/output procedure (9.6.4.8), or a NEWUNIT value (9.5.6.12). 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.

3 The external unit identified by a particular value of a *scalar-int-expr* is the same external unit in all program units of the program.

NOTE 9.13

In the example:

```
SUBROUTINE A
   READ (6) X
      ...
SUBROUTINE B
   N = 6
   REWIND N
```

the value 6 used in both program units identifies the same external unit.

4 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 only (9.6.4.3). This unit is also identified by the value of the named constant INPUT_UNIT of the intrinsic module ISO_FORTRAN_ENV (13.8.2.10). 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 (13.8.2.19).

5 This part of ISO/IEC 1539 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 9.14

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.

9.5.2 Connection modes

1 A connection for formatted input/output has several changeable modes: these are the blank interpretation mode (10.8.6), delimiter mode (10.10.4, 10.11.4.2), sign mode (10.8.4), decimal edit mode (10.8.8), input/output rounding mode (10.7.2.3.8), pad mode (9.6.4.5.3), and scale factor (10.8.5). A connection for unformatted input/output has no changeable modes.

2 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.

3 The scale factor cannot be explicitly specified in an OPEN statement; it is implicitly 0.

4 The modes of a connection to an external file may be changed by a subsequent OPEN statement that modifies the connection.

5 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.

9.5.3 Unit existence

1 At any given time, there is a processor-dependent set of external units that exist for an image.

2 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.

9.5.4 Connection of a file to a unit

1 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.

2 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.

3 A file may be connected and not exist (9.3.2).

NOTE 9.15

An example is a preconnected external file that has not yet been written.

4 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.

5 This part of ISO/IEC 1539 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.

6 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 9.16

> 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.

7 An internal unit is always connected to the internal file designated by the variable that identifies the unit.

NOTE 9.17

> For more explanatory information on file connection properties, see C.6.4.

9.5.5 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.

9.5.6 OPEN statement

9.5.6.1 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 and, once connected, a reference to it may appear in any program unit of the program.

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 (9.5.2) 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.

8 If a file is already connected to a unit, an OPEN statement on that file with a different unit shall not be executed.

9.5.6.2 Syntax

R904	*open-stmt*	**is**	OPEN (*connect-spec-list*)
R905	*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*

R906 *file-name-expr* **is** *scalar-default-char-expr*

R907 *iomsg-variable* **is** *scalar-default-char-variable*

C903 No specifier shall appear more than once in a given *connect-spec-list*.

C904 (R904) 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*.

C905 (R904) 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.

C906 (R904) If a NEWUNIT= specifier appears, a *file-unit-number* shall not appear.

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 9.11.

NOTE 9.18

An example of an OPEN statement is:

```
OPEN (10, FILE = 'employee.names', ACTION = 'READ', PAD = 'YES')
```

NOTE 9.19

For more explanatory information on the OPEN statement, see C.6.3.

9.5.6.3 ACCESS= specifier in the OPEN statement

1 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.

9.5.6.4 ACTION= specifier in the OPEN statement

1 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.

9.5.6.5 ASYNCHRONOUS= specifier in the OPEN statement

1 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.

9.5.6.6 BLANK= specifier in the OPEN statement

1 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 (10.8.6, 9.6.2.6) for input for this connection. This mode has no effect on output. It is a changeable mode (9.5.2). If this specifier is omitted in an OPEN statement that initiates a connection, the default value is NULL.

9.5.6.7 DECIMAL= specifier in the OPEN statement

1 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 (10.6, 10.8.8, 9.6.2.7) for this connection. This is a changeable mode (9.5.2). If this specifier is omitted in an OPEN statement that initiates a connection, the default value is POINT.

9.5.6.8 DELIM= specifier in the OPEN statement

1 The *scalar-default-char-expr* shall evaluate to APOSTROPHE, QUOTE, or NONE. The DELIM= specifier is permitted only for a connection for formatted input/output. It specifies the delimiter mode (9.6.2.8) for list-directed (10.10.4) and namelist (10.11.4.2) output for the connection. This mode has no effect on input. It is a changeable mode (9.5.2). If this specifier is omitted in an OPEN statement that initiates a connection, the default value is NONE.

9.5.6.9 ENCODING= specifier in the OPEN statement

1 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. The value UTF-8 specifies that 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.

9.5.6.10 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.

9.5.6.11 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.

9.5.6.12 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 (13.8.2), 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.

9.5.6.13 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 (9.6.4.5.3, 9.6.2.10) for input for this connection. This mode has no effect on output. It is a changeable mode (9.5.2). If this specifier is omitted in an OPEN statement that initiates a connection, the default value is YES.

9.5.6.14 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.

9.5.6.15 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.

9.5.6.16 ROUND= specifier in the OPEN statement

1 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 (10.7.2.3.8, 9.6.2.13) for this connection. This is a changeable mode (9.5.2). 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 9.20

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.

9.5.6.17 SIGN= specifier in the OPEN statement

1 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 (10.8.4, 9.6.2.14) for this connection. This is a changeable mode (9.5.2). If this specifier is omitted in an OPEN statement that initiates a connection, the default value is PROCESSOR_DEFINED.

9.5.6.18 STATUS= specifier in the OPEN statement

1 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.

2 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.

NOTE 9.21

SCRATCH shall not be specified with a named file.

3 If UNKNOWN is specified, the status is processor dependent. If this specifier is omitted, the default value is UNKNOWN.

9.5.7 CLOSE statement

9.5.7.1 General

1 The CLOSE statement is used to terminate the connection of a specified unit to an external file.

2 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.

3 Execution of a CLOSE statement performs a wait operation for any pending asynchronous data transfer operations for the specified unit.

4 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.

5 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.

6 During the completion step (2.3.6) of termination of execution of a program, 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 9.22

> The effect is as though a CLOSE statement without a STATUS= specifier were executed on each connected unit.

9.5.7.2 Syntax

R908	*close-stmt*	**is**	CLOSE (*close-spec-list*)
R909	*close-spec*	**is**	[UNIT =] *file-unit-number*
		or	IOSTAT = *scalar-int-variable*
		or	IOMSG = *iomsg-variable*
		or	ERR = *label*
		or	STATUS = *scalar-default-char-expr*

C907 No specifier shall appear more than once in a given *close-spec-list*.

C908 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*.

C909 (R909) 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 9.11.

NOTE 9.23

> An example of a CLOSE statement is:
>
> ```
> CLOSE (10, STATUS = 'KEEP')
> ```

9.5.7.3 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.

9.6 Data transfer statements

9.6.1 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.

R910	*read-stmt*	**is**	READ (*io-control-spec-list*) [*input-item-list*]
		or	READ *format* [, *input-item-list*]
R911	*write-stmt*	**is**	WRITE (*io-control-spec-list*) [*output-item-list*]

R912 *print-stmt* **is** PRINT *format* [, *output-item-list*]

NOTE 9.24

Examples of data transfer statements are:

```
   READ (6, *) SIZE
   READ 10, A, B
   WRITE (6, 10) A, S, J
   PRINT 10, A, S, J
10 FORMAT (2E16.3, I5)
```

9.6.2 Control information list

9.6.2.1 Syntax

1 A control information list is an *io-control-spec-list*. It governs data transfer.

R913 *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*

R914 *id-variable* **is** *scalar-int-variable*

C910 No specifier shall appear more than once in a given *io-control-spec-list*.

C911 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*.

C912 (R913) A DELIM= or SIGN= specifier shall not appear in a *read-stmt*.

C913 (R913) A BLANK=, PAD=, END=, EOR=, or SIZE= specifier shall not appear in a *write-stmt*.

C914 (R913) 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.

C915 (R913) A *namelist-group-name* shall be the name of a namelist group.

C916 (R913) 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.

C917 (R913) If *format* appears without a preceding FMT=, it shall be the second item in the *io-control-spec-list* and the first item shall be *io-unit*.

C918 (R913) If *namelist-group-name* appears without a preceding NML=, it shall be the second item in the *io-control-spec-list* and the first item shall be *io-unit*.

C919 (R913) If *io-unit* is not a *file-unit-number*, the *io-control-spec-list* shall not contain a REC= specifier or a POS= specifier.

C920 (R913) If the REC= specifier appears, an END= specifier shall not appear, and the *format*, if any, shall not be an asterisk.

C921 (R913) An ADVANCE= specifier may appear only in a formatted sequential or stream data transfer statement with explicit format specification (10.2) whose *io-control-spec-list* does not contain an *internal-file-variable* as the *io-unit*.

C922 (R913) If an EOR= specifier appears, an ADVANCE= specifier also shall appear.

C923 (R913) The *scalar-default-char-constant-expr* in an ASYNCHRONOUS= specifier shall have the value YES or NO.

C924 (R913) An ASYNCHRONOUS= specifier with a value YES shall not appear unless *io-unit* is a *file-unit-number*.

C925 (R913) If an ID= specifier appears, an ASYNCHRONOUS= specifier with the value YES shall also appear.

C926 (R913) If a POS= specifier appears, the *io-control-spec-list* shall not contain a REC= specifier.

C927 (R913) If a DECIMAL=, BLANK=, PAD=, SIGN=, or ROUND= specifier appears, a *format* or *namelist-group-name* shall also appear.

C928 (R913) If a DELIM= specifier appears, either *format* shall be an asterisk or *namelist-group-name* shall appear.

C929 (R914) The *scalar-int-variable* shall have a decimal exponent range no smaller than that of default integer.

2 If an EOR= specifier appears, an ADVANCE= specifier with the value NO shall also appear.

3 If the data transfer statement contains a *format* or *namelist-group-name*, the statement is a formatted input/output statement; otherwise, it is an unformatted input/output statement.

4 The ADVANCE=, ASYNCHRONOUS=, DECIMAL=, BLANK=, DELIM=, 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.

5 The IOSTAT=, ERR=, EOR=, END=, and IOMSG= specifiers are described in 9.11.

NOTE 9.25

An example of a READ statement is:

```
READ (IOSTAT = IOS, UNIT = 6, FMT = '(10F8.2)') A, B
```

9.6.2.2 Format specification in a data transfer statement

1 The *format* specifier supplies a format specification or specifies list-directed formatting for a formatted input/output statement.

R915 *format* **is** *default-char-expr*
or *label*

or *

C930 (R915) 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 (10.2.1 and 10.2.2).

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 9.26

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.

9.6.2.3 NML= specifier in a data transfer statement

1 The NML= specifier supplies the *namelist-group-name* (5.8). 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.

9.6.2.4 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 (9.3.4.2). 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.

9.6.2.5 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 9.27

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.

4 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.

5 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 9.28

A pending input/output storage sequence is not necessarily a contiguous set of storage units.

6 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.

9.6.2.6 BLANK= specifier in a data transfer statement

1 The *scalar-default-char-expr* shall evaluate to NULL or ZERO. The BLANK= specifier temporarily changes (9.5.2) the blank interpretation mode (10.8.6, 9.5.6.6) for the connection. If the specifier is omitted, the mode is not changed.

9.6.2.7 DECIMAL= specifier in a data transfer statement

1 The *scalar-default-char-expr* shall evaluate to COMMA or POINT. The DECIMAL= specifier temporarily changes (9.5.2) the decimal edit mode (10.6, 10.8.8, 9.5.6.7) for the connection. If the specifier is omitted, the mode is not changed.

9.6.2.8 DELIM= specifier in a data transfer statement

1 The *scalar-default-char-expr* shall evaluate to APOSTROPHE, QUOTE, or NONE. The DELIM= specifier temporarily changes (9.5.2) the delimiter mode (10.10.4, 10.11.4.2, 9.5.6.8) for the connection. If the specifier is omitted, the mode is not changed.

9.6.2.9 ID= specifier in a data transfer statement

1 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.

2 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.

3 A child data transfer statement shall not specify the ID= specifier.

9.6.2.10 PAD= specifier in a data transfer statement

1 The *scalar-default-char-expr* shall evaluate to YES or NO. The PAD= specifier temporarily changes (9.5.2) the pad mode (9.6.4.5.3, 9.5.6.13) for the connection. If the specifier is omitted, the mode is not changed.

9.6.2.11 POS= specifier in a data transfer statement

1 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.

2 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 9.29

A unit that is connected to a device or data stream might not be positionable.

3 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.

9.6.2.12 REC= specifier in a data transfer statement

1 The REC= specifier specifies the number of the record that is to be read or written. This specifier may appear only in an 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.

9.6.2.13 ROUND= specifier in a data transfer statement

1 The *scalar-default-char-expr* shall evaluate to UP, DOWN, ZERO, NEAREST, COMPATIBLE or PROCESSOR_DEFINED. The ROUND= specifier temporarily changes (9.5.2) the input/output rounding mode (10.7.2.3.8, 9.5.6.16) for the connection. If the specifier is omitted, the mode is not changed.

9.6.2.14 SIGN= specifier in a data transfer statement

1 The *scalar-default-char-expr* shall evaluate to PLUS, SUPPRESS, or PROCESSOR_DEFINED. The SIGN= specifier temporarily changes (9.5.2) the sign mode (10.8.4, 9.5.6.17) for the connection. If the specifier is omitted, the mode is not changed.

9.6.2.15 SIZE= specifier in a data transfer statement

1 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.

2 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.

9.6.3 Data transfer input/output list

1 An input/output list specifies the entities whose values are transferred by a data transfer statement.

R916	*input-item*	**is**	*variable*
		or	*io-implied-do*
R917	*output-item*	**is**	*expr*
		or	*io-implied-do*

R918 *io-implied-do* **is** (*io-implied-do-object-list* , *io-implied-do-control*)

R919 *io-implied-do-object* **is** *input-item*
or *output-item*

R920 *io-implied-do-control* **is** *do-variable* = *scalar-int-expr* , ■
■ *scalar-int-expr* [, *scalar-int-expr*]

C931 (R916) A variable that is an *input-item* shall not be a whole assumed-size array.

C932 (R919) 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*.

C933 (R917) An expression that is an *output-item* shall not have a value that is a procedure pointer.

2 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 9.30

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 9.31

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 (9.6.4.8).

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 (6.5.3.2). 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 9.32

For example:

```
INTEGER A (100), J (100)
   ...
READ *, A (A)                            ! Not allowed
READ *, A (LBOUND (A, 1) : UBOUND (A, 1))  ! Allowed
READ *, A (J)                            ! Allowed if no two elements
                                         !    of J have the same value
A(1) = 1; A(10) = 10
READ *, A (A (1) : A (10))               ! Not allowed
```

- If a list item of derived type in an unformatted input/output statement is not processed by a defined input/output procedure (9.6.4.8), 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 (4.5.4.7); 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 (9.6.4.8).

NOTE 9.33

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 (8.1.6.4).

NOTE 9.34

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
```

8 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-do*s 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 9.35

In a formatted input/output statement, edit descriptors are associated with effective items, which are always scalar. The rules in 9.6.3 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.

9 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.

9.6.4 Execution of a data transfer input/output statement

9.6.4.1 Data transfer sequence of operations

1 Execution of a WRITE or PRINT statement for 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.
(2) Identify the unit.
(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 (9.6.4.8),
    (a) position the file prior to data transfer (9.3.4.3), and
    (b) for formatted data transfer, set the left tab limit (10.8.1.1).
(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 (9.6.4.8).
(7) Determine whether an error, end-of-file, or end-of-record condition has occurred.
(8) Position the file after data transfer (9.3.4.4) unless the statement is a child data transfer statement (9.6.4.8).
(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 9.11; 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.
(2) Identify the unit.
(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 (9.3.4.3) and, for formatted data transfer, set the left tab limit (10.8.1.1).
(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 (9.3.4.4).
(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 9.11; otherwise any variable specified in an IOSTAT= specifier is assigned the value zero.

4 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.

5 For asynchronous output, a pending input/output storage sequence affector (9.6.2.5) shall not be redefined, become undefined, or have its pointer association status changed.

6 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.

7 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.

8 If execution of the program is terminated during execution of an output statement, the contents of the file become undefined.

NOTE 9.36

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.

9.6.4.2 Direction of data transfer

1 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.

9.6.4.3 Identifying a unit

1 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 (9.5.1) and is the same as the unit identified by the value of the named constant INPUT_UNIT of the intrinsic module ISO_FORTRAN_ENV (13.8.2.10). 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 (13.8.2.19). Thus, each data transfer statement identifies an external or internal unit.

2 The unit identified by an unformatted data transfer statement shall be an external unit.

3 The unit identified by a data transfer statement shall be connected to a file when execution of the statement begins.

NOTE 9.37

The unit could be preconnected.

9.6.4.4 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.

9.6.4.5 Data transfer

9.6.4.5.1 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 9.6.3.

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 9.38

In the example

```
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 (10.11) 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.

9.6.4.5.2 Unformatted data transfer

1 If the file is not connected for unformatted input/output, unformatted data transfer is prohibited.

2 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.

3 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.

4 After each value is transferred, the current file position is moved to a point immediately after the last file storage unit of the value.

5 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.

6 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.

7 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.

8 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.

9.6.4.5.3 Formatted data transfer

1 If the file is not connected for formatted input/output, formatted data transfer is prohibited.

2 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 10.

3 The current record and possibly additional records are read or written.

4 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.

5 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.

6 During nonadvancing input when the pad mode has the value NO, an end-of-record condition (9.11) occurs if the input list and format specification require more characters from the record than the record contains, and the record is complete (9.3.3.4). If the record is incomplete, an end-of-file condition occurs instead of an end-of-record condition.

7 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-file condition occurs; otherwise an end-of-record condition

occurs.

8 If the file is connected for direct access, the record number is increased by one as each succeeding record is read or written.

9 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.

10 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.

9.6.4.6 List-directed formatting

1 If list-directed formatting has been established, editing is performed as described in 10.10.

9.6.4.7 Namelist formatting

1 If namelist formatting has been established, editing is performed as described in 10.11.

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 (9.6.4.8).

9.6.4.8 Defined input/output

9.6.4.8.1 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 9.6.3.

2 A defined input/output procedure is a procedure accessible by a *defined-io-generic-spec* (12.4.3.2). A particular defined input/output procedure is selected as described in 9.6.4.8.4.

9.6.4.8.2 Executing defined input/output data transfers

1 If a defined input/output procedure is selected for an effective item as specified in 9.6.4.8.4, 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 9.39

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 9.40

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,

NOTE 9.40 (cont.)

> 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.

9.6.4.8.3 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 user need not supply all four procedures. The procedures are specified to be used for derived-type input/output by interface blocks (12.4.3.2) or by generic bindings (4.5.5), with a *defined-io-generic-spec* (R1209). The *defined-io-generic-spec*s for these procedures are READ (FORMATTED), READ (UNFORMATTED), WRITE (FORMATTED), and WRITE (UNFORMATTED), for formatted input, unformatted 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:

R921	*dtv-type-spec*	**is**	TYPE(*derived-type-spec*)
		or	CLASS(*derived-type-spec*)

C934 (R921) If *derived-type-spec* specifies an extensible type, the CLASS keyword shall be used; otherwise, the TYPE keyword shall be used.

C935 (R921) 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:

4
```
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:

6
```
SUBROUTINE my_read_routine_unformatted                 &
                              (dtv,                     &
```

```
                                     unit,                   &
                                     iostat, iomsg)
          ! the derived-type variable
          dtv-type-spec , INTENT(INOUT) :: dtv
          INTEGER, INTENT(IN) :: unit
          INTEGER, INTENT(OUT) :: iostat
          CHARACTER (LEN=*), INTENT(INOUT) :: iomsg
        END
```

7 If the *defined-io-generic-spec* is WRITE (FORMATTED), the characteristics shall be the same as those specified by the following interface:

8
```
        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
```

9 If the *defined-io-generic-spec* is WRITE (UNFORMATTED), the characteristics shall be the same as those specified by the following interface:

10
```
        SUBROUTINE my_write_routine_unformatted              &
                                    (dtv,                    &
                                     unit,                   &
                                     iostat, iomsg)
          ! the derived-type value/variable
          dtv-type-spec , INTENT(IN) :: dtv
          INTEGER, INTENT(IN) :: unit
          INTEGER, INTENT(OUT) :: iostat
          CHARACTER (LEN=*), INTENT(INOUT) :: iomsg
        END
```

11 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 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(13.8.2).

- 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(13.8.2).
- Otherwise the parent data transfer statement must access an internal file, in which case the `unit` argument shall have a processor-dependent negative value.

NOTE 9.41

> 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(13.8.2).

13 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.

14 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 (2.4.5).

15 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.

16 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 9.42

> 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.

17 The `iostat` argument is used to report whether an error, end-of-record, or end-of-file condition (9.11) 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 (13.8.2.13) 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 (13.8.2.14) to `iostat`. Otherwise, the defined input/output procedure shall assign the value zero to the `iostat` argument.

18 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 9.43

> 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.

19 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 9.11, the processor shall make the value of the `iomsg` argument available in a processor-dependent manner.

20 When 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.

21 When 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.

22 When a parent data transfer statement is active, a data transfer statement that specifies an internal file is permitted.

23 OPEN, CLOSE, BACKSPACE, ENDFILE, and REWIND statements shall not be executed while a parent data transfer statement is active.

24 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.

25 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.

26 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.

NOTE 9.44

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 9.10.

27 Neither a parent nor child data transfer statement shall be asynchronous.

28 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.

NOTE 9.45

A child data transfer statement shall not specify the ID=, POS=, or REC= specifiers in an input/output control list.

NOTE 9.46

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
  CONTAINS
    PROCEDURE,PRIVATE :: pwf
    GENERIC           :: WRITE(FORMATTED) => pwf
  END TYPE person

CONTAINS
```

NOTE 9.46 (cont.)

```
  SUBROUTINE pwf (dtv,unit,iotype,vlist,iostat,iomsg)
! argument definitions
    CLASS(person), INTENT(IN) :: dtv
    INTEGER, INTENT(IN) :: unit
    CHARACTER (LEN=*), INTENT(IN) :: iotype
    INTEGER, INTENT(IN) :: vlist(:)
    INTEGER, INTENT(OUT) :: iostat
    CHARACTER (LEN=*), INTENT(INOUT) :: 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 9.47

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
```

NOTE 9.47 (cont.)

```
    INTEGER, INTENT(IN) :: unit
    CHARACTER (LEN=*), INTENT(IN) :: iotype
    INTEGER, INTENT(IN) :: vlist(:)
    INTEGER, INTENT(OUT) :: iostat
    CHARACTER (LEN=*), INTENT(INOUT) :: iomsg

    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
```

9.6.4.8.4 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 9.6.4.8.3, and has a specific interface whose `dtv` argument is compatible with the effective item according to the rules for argument association in 12.5.2.4.

2 When an effective item (9.6.3) 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 (12.5); 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.

9.6.5 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 (10.10, 10.11) 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 (9.11).

9.7 Waiting on pending data transfer

9.7.1 Wait operation

1 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.

2 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.

3 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 9.6.2.15 and 9.6.4.5.

4 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.

5 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 9.48

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.

6 After completion of the wait operation, the data transfer operation and its input/output storage sequence are no longer considered to be pending.

9.7.2 WAIT statement

1 A WAIT statement performs a wait operation for specified pending asynchronous data transfer operations.

R922 *wait-stmt* **is** WAIT (*wait-spec-list*)

R923 *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*

C936 No specifier shall appear more than once in a given *wait-spec-list*.

C937 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*.

C938 (R923) 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.

2 The IOSTAT=, ERR=, EOR=, END=, and IOMSG= specifiers are described in 9.11.

3 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.

4 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 9.49

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.

9.8 File positioning statements

9.8.1 Syntax

R924 *backspace-stmt* **is** BACKSPACE *file-unit-number*
or BACKSPACE (*position-spec-list*)

R925 *endfile-stmt* **is** ENDFILE *file-unit-number*
or ENDFILE (*position-spec-list*)

R926 *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.

R927 *position-spec* **is** [UNIT =] *file-unit-number*
or IOMSG = *iomsg-variable*
or IOSTAT = *scalar-int-variable*
or ERR = *label*

C939 No specifier shall appear more than once in a given *position-spec-list*.

C940 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*.

C941 (R927) 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.

2 The IOSTAT=, ERR=, and IOMSG= specifiers are described in 9.11.

3 Execution of a file positioning statement performs a wait operation for all pending asynchronous data transfer operations for the specified unit.

9.8.2 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 9.50

If the preceding record is an endfile record, the file is positioned before the endfile record.

2 If a BACKSPACE statement causes the implicit writing of an endfile record, the file is positioned before the record that precedes the endfile record.

3 Backspacing a file that is connected but does not exist is prohibited.

4 Backspacing over records written using list-directed or namelist formatting is prohibited.

NOTE 9.51

An example of a BACKSPACE statement is:

```
BACKSPACE (10, IOSTAT = N)
```

9.8.3 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 9.52

An example of an ENDFILE statement is:

```
ENDFILE K
```

9.8.4 REWIND statement

1 Execution of a REWIND statement causes the specified file to be positioned at its initial point.

NOTE 9.53

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 9.54

An example of a REWIND statement is:

```
REWIND 10
```

9.9 FLUSH statement

R928 *flush-stmt* **is** FLUSH *file-unit-number*
or FLUSH (*flush-spec-list*)

R929 *flush-spec* **is** [UNIT =] *file-unit-number*
or IOSTAT = *scalar-int-variable*
or IOMSG = *iomsg-variable*
or ERR = *label*

C942 No specifier shall appear more than once in a given *flush-spec-list*.

C943 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*.

C944 (R929) 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.

1 The IOSTAT=, IOMSG= and ERR= specifiers are described in 9.11. The IOSTAT= variable shall be set to a processor-dependent positive value if an error occurs, to zero if the processor-dependent flush operation was successful, or to a processor-dependent negative value if the flush operation is not supported for the unit specified.

2 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.

3 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.

4 Execution of a FLUSH statement performs a wait operation for all pending asynchronous data transfer operations for the specified unit.

NOTE 9.55

Because this part of ISO/IEC 1539 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 9.56

An example of a FLUSH statement is:

```
FLUSH (10, IOSTAT = N)
```

9.10 File inquiry statement

9.10.1 Forms of the INQUIRE statement

1 The INQUIRE statement may be used to inquire about properties of a particular named file or of the connection to a particular unit. 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. All specifier value assignments are performed as if by intrinsic assignment.

2 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.

3 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.

R930 *inquire-stmt* **is** INQUIRE (*inquire-spec-list*)
or INQUIRE (IOLENGTH = *scalar-int-variable*) ■
■ *output-item-list*

NOTE 9.57

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)
```

9.10.2 Inquiry specifiers

9.10.2.1 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.

R931 *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*

C945 No specifier shall appear more than once in a given *inquire-spec-list*.

C946 An *inquire-spec-list* shall contain one FILE= specifier or one *file-unit-number*, but not both.

C947 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*.

C948 If an ID= specifier appears in an *inquire-spec-list*, a PENDING= specifier shall also appear.

C949 (R929) 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 (9.6.4.8.3), 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 9.11.

9.10.2.2 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.

9.10.2.3 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.

9.10.2.4 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.

9.10.2.5 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.

9.10.2.6 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.

9.10.2.7 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.

9.10.2.8 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.

9.10.2.9 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.

9.10.2.10 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 9.58

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).

9.10.2.11 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.

9.10.2.12 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.

9.10.2.13 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.

9.10.2.14 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 (9.10.2.21).

9.10.2.15 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 9.59

> 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.

9.10.2.16 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.

9.10.2.17 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.

9.10.2.18 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 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*.

9.10.2.19 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.

9.10.2.20 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.

9.10.2.21 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.

2 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.

3 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.

4 In all other cases, the variable specified in the PENDING= specifier is assigned the value true and no wait operations are performed; in this case the previously pending data transfers remain pending after the execution of the INQUIRE statement.

NOTE 9.60

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.

9.10.2.22 POS= specifier in the INQUIRE statement

1 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 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.

9.10.2.23 POSITION= specifier in the INQUIRE statement

1 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.

9.10.2.24 READ= specifier in the INQUIRE statement

1 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.

9.10.2.25 READWRITE= specifier in the INQUIRE statement

1 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.

9.10.2.26 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.

9.10.2.27 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.

9.10.2.28 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.

9.10.2.29 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.

9.10.2.30 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.

9.10.2.31 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.

9.10.2.32 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.

9.10.2.33 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.

9.10.3 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 when there are data transfer statements with the same input/output list.

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 9.6.3. 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.

9.11 Error, end-of-record, and end-of-file conditions

9.11.1 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.

9.11.2 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-variable*s 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 9.11.5;

(4) if an IOMSG= specifier appears, the *iomsg-variable* becomes defined as specified in 9.11.6;

(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 9.6.2.15;

(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.

9.11.3 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-variable*s 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 9.11.5;

(6) if an IOMSG= specifier appears, the *iomsg-variable* becomes defined as specified in 9.11.6;

(7) if an END= specifier appears, a branch to the statement labeled by the *label* in the END= specifier occurs.

9.11.4 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 (9.6.4.5.3) 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-variable*s 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 9.11.5;

(6) if an IOMSG= specifier appears, the *iomsg-variable* becomes defined as specified in 9.11.6;

(7) if a SIZE= specifier appears, the *scalar-int-variable* in the SIZE= specifier becomes defined as specified in (9.6.2.15);

(8) if an EOR= specifier appears, a branch to the statement labeled by the *label* in the EOR= specifier occurs.

9.11.5 IOSTAT= specifier

1 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(13.8.2) 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 (13.8.2.13) from the intrinsic module ISO_FORTRAN_ENV if an end-of-file condition occurs and no error condition occurs, or
- the processor-dependent negative integer value of the constant IOSTAT_EOR (13.8.2.14) from the intrinsic module ISO_FORTRAN_ENV if an end-of-record condition occurs and no error condition or end-of-file condition occurs.

NOTE 9.61

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,

```
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
```

9.11.6 IOMSG= specifier

1 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.

9.12 Restrictions on input/output statements

1 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.

2 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.

3 An input/output statement shall not cause the value of any established format specification to be modified.

4 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.

5 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 values of any *input-item* or *io-implied-do do-variable* in the same statement.

6 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.

7 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 (9.6.3) or *namelist-group-object-list*, nor with a *do-variable* of an *io-implied-do* in the data transfer input/output list.

8 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*.

9 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 9.62

Restrictions on the evaluation of expressions (7.1.4) prohibit certain side effects.

(Blank page)

10 Input/output editing

10.1 Format specifications

1 A format used in conjunction with 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* (9.6.2.2) 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 (10.10). Namelist formatting (10.11) may be indicated by specifying a *namelist-group-name* instead of a *format*.

10.2 Explicit format specification methods

10.2.1 FORMAT statement

R1001 *format-stmt* **is** FORMAT *format-specification*

R1002 *format-specification* **is** ([*format-items*])
or ([*format-items*,] *unlimited-format-item*)

C1001 (R1001) The *format-stmt* shall be labeled.

1 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 (10.9).

NOTE 10.1

Examples of FORMAT statements are:

```
5     FORMAT (1PE12.4, I10)
9     FORMAT (I12, /, ' Dates: ', 2 (2I3, I5))
```

10.2.2 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 10.2

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 10.3

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.

10.3 Form of a format item list

10.3.1 Syntax

R1003	*format-items*	**is**	*format-item* [[,] *format-item*] ...
R1004	*format-item*	**is**	[*r*] *data-edit-desc*
		or	*control-edit-desc*
		or	*char-string-edit-desc*
		or	[*r*] (*format-items*)
R1005	*unlimited-format-item*	**is**	* (*format-items*)
R1006	*r*	**is**	*int-literal-constant*

C1002 (R1003) 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 (10.8.5), possibly preceded by a repeat specification,
- before a slash edit descriptor when the optional repeat specification does not appear (10.8.2),
- after a slash edit descriptor, or
- before or after a colon edit descriptor (10.8.3)

C1003 (R1005) An *unlimited-format-item* shall contain at least one data edit descriptor.

C1004 (R1006) *r* shall be positive.

C1005 (R1006) A kind parameter shall not be specified for *r*.

1 The integer literal constant *r* is called a repeat specification.

10.3.2 Edit descriptors

1 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*).

R1007 *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*)]

R1008 *w* **is** *int-literal-constant*

R1009 *m* **is** *int-literal-constant*

R1010 *d* **is** *int-literal-constant*

R1011 *e* **is** *int-literal-constant*

R1012 *v* **is** *signed-int-literal-constant*

C1006 (R1008) *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.

C1007 (R1007) For the G edit descriptor, *d* shall be specified if *w* is not zero.

C1008 (R1007) For the G edit descriptor, *e* shall not be specified if *w* is zero.

C1009 (R1007) 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*.

2 I, B, O, Z, F, E, EN, ES, EX, G, L, A, D, and DT indicate the manner of editing.

R1013 *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*

R1014 *k* **is** *signed-int-literal-constant*

C1010 (R1014) A kind parameter shall not be specified for *k*.

3 In *k* P, *k* is called the scale factor.

R1015 *position-edit-desc* **is** T *n*
or TL *n*
or TR *n*
or *n* X

R1016 *n* **is** *int-literal-constant*

C1011 (R1016) *n* shall be positive.

C1012 (R1016) A kind parameter shall not be specified for *n*.

R1017 *sign-edit-desc* **is** SS
or SP
or S

R1018 *blank-interp-edit-desc* **is** BN
or BZ

R1019 *round-edit-desc* **is** RU
or RD
or RZ
or RN
or RC
or RP

R1020 *decimal-edit-desc* **is** DC
or DP

4 T, TL, TR, X, slash, colon, SS, SP, S, P, BN, BZ, RU, RD, RZ, RN, RC, RP, DC, and DP indicate the manner of editing.

R1021 *char-string-edit-desc* **is** *char-literal-constant*

C1013 (R1021) A kind parameter shall not be specified for the *char-literal-constant*.

5 Each *rep-char* in a character string edit descriptor shall be one of the characters capable of representation by the processor.

6 The character string edit descriptors provide constant data to be output, and are not valid for input.

7 The edit descriptors are without regard to case except for the characters in the character constants.

10.3.3 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.

10.4 Interaction between input/output list and format

1 The start of formatted data transfer using a format specification initiates format control (9.6.4.5.3). Each action of format control depends on information jointly provided by the next edit descriptor in the format specification and the next effective item in the input/output list, if one exists.

2 If an input/output list specifies at least one effective item, at least one data edit descriptor shall exist in the format specification.

NOTE 10.4

An empty format specification of the form ( ) can be used only if the input/output list has no effective item (9.6.4.5). A zero length character item is an 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 10.5

An omitted repeat specification is treated in the same way as a repeat specification whose value is one.

5 To each data edit descriptor interpreted in a format specification, there corresponds one effective item specified by the input/output list (9.6.3), 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.

6 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.

7 If format control encounters a colon edit descriptor in a format specification and another effective item is not specified, format control terminates.

8 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 (9.5.2).

9 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 (10.8.2).

NOTE 10.6

Example: The format specification:

```
10 FORMAT (1X, 2(F10.3, I5))
```

with an output list of

```
WRITE (10,10) 10.1, 3, 4.7, 1, 12.4, 5, 5.2, 6
```

produces the same output as the format specification:

```
10 FORMAT (1X, F10.3, I5, F10.3, I5/F10.3, I5, F10.3, I5)
```

NOTE 10.7

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,

```
WRITE( 10, '( "IARRAY =", *( I0, :, ","))') IARRAY
```

produces a single record with a header and a comma separated list of integer values.

10.5 Positioning by format control

1 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.

2 After each T, TL, TR, or X edit descriptor is processed, the file is positioned as described in 10.8.1. After each slash edit descriptor is processed, the file is positioned as described in 10.8.2.

3 During formatted stream output, processing of an A edit descriptor can cause file positioning to occur (10.7.4).

4 If format control reverts as described in 10.4, the file is positioned in a manner identical to the way it is positioned when a slash edit descriptor is processed (10.8.2).

5 During a read operation, any unprocessed characters of the current record are skipped whenever the next record is read.

10.6 Decimal symbol

1 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.

2 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.

10.7 Data edit descriptors

10.7.1 Purpose of data edit descriptors

1 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.

2 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.

3 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.

4 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.

5 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.

10.7.2 Numeric editing

10.7.2.1 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 (10.7.2.3.2), the interpretation of blanks, other than leading blanks, is determined by the blank interpretation mode (10.8.6). 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 10.8

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.

10.7.2.2 Integer editing

1 The Iw 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 (10.7.5.2.1, 10.7.2.4).

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* (R410), 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 Iw 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 10.9

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 Iw 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.

10.7.2.3 Real and complex editing

10.7.2.3.1 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 (10.7.5.2.2, 10.7.2.4).

10.7.2.3.2 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-significant 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 10.10

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 (10.8.5).

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.

10 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 10.11

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 (10.7.2.3.8) 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.

10.7.2.3.3 E and D editing

1 The E$w.d$, D$w.d$, and E$w.d$ Ee 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 (10.7.2.3.2).

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

$[\pm]\ [0].x_1x_2\ldots x_d exp$

where:

- $\pm$ signifies a plus sign or a minus sign;
- . signifies a decimal symbol (10.6);
- $x_1x_2\ldots x_d$ are the d most significant digits of the internal value after rounding (10.7.2.3.8);
- *exp* is a decimal exponent having one of the forms specified in table 10.1.

Table 10.1: **E and D exponent forms**

Edit Descriptor	Absolute Value of Exponent	Form of Exponent[1]
E$w.d$	$\|exp\| \leq 99$	E$\pm z_1 z_2$ or $\pm 0 z_1 z_2$
	$99 < \|exp\| \leq 999$	$\pm z_1 z_2 z_3$
E$w.d$ Ee with $e > 0$	$\|exp\| \leq 10^e - 1$	E$\pm z_1 z_2 \ldots z_e$
E$w.d$ E0	any	E$\pm z_1 z_2 \ldots z_s$
D$w.d$	$\|exp\| \leq 99$	D$\pm z_1 z_2$ or E$\pm z_1 z_2$ or $\pm 0 z_1 z_2$
	$99 < \|exp\| \leq 999$	$\pm z_1 z_2 z_3$
(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 (10.3.2, 10.8.5). 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.

10.7.2.3.4 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 (R415) 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$w.d$ and EN$w.d$ Ee 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 (10.7.2.3.2).

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

$[\pm]\ yyy\ .\ x_1 x_2 \ldots x_d exp$

where:

- $\pm$ signifies a plus sign or a minus sign;
- yyy are the 1 to 3 decimal digits representative of the most significant digits of the internal value after rounding (10.7.2.3.8);
- yyy is an integer such that $1 \leq yyy < 1000$ or, if the output value is zero, $yyy = 0$;
- . signifies a decimal symbol (10.6);
- $x_1 x_2 \ldots x_d$ are the d next most significant digits of the internal value after rounding;
- exp is a decimal exponent, divisible by three, having one of the forms specified in table 10.2.

Table 10.2: **EN exponent forms**

Edit Descriptor	Absolute Value of Exponent	Form of Exponent[1]
EN$w.d$	$\|exp\| \leq 99$	E$\pm z_1 z_2$ or $\pm 0 z_1 z_2$
	$99 < \|exp\| \leq 999$	$\pm z_1 z_2 z_3$
EN$w.d$ Ee with $e > 0$	$\|exp\| \leq 10^e - 1$	E$\pm z_1 z_2 \ldots z_e$
EN$w.d$ E0	any	E$\pm z_1 z_2 \ldots z_s$

EN exponent forms (cont.)

Edit Descriptor	Absolute Value of Exponent	Form of Exponent[1]
(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 10.12

Examples:

Internal Value	Output field Using SS, EN12.3
`6.421`	`6.421E+00`
`-.5`	`-500.000E-03`
`.00217`	`2.170E-03`
`4721.3`	`4.721E+03`

10.7.2.3.5 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 (R415) 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$w.d$ Ee 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 (10.7.2.3.2).

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

$[\pm]\ y\ .\ x_1x_2 \ldots x_d exp$

where:

- $\pm$ 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 (10.7.2.3.8);
- . signifies a decimal symbol (10.6);
- $x_1x_2 \ldots x_d$ 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 10.3.

Table 10.3: **ES exponent forms**

Edit Descriptor	Absolute Value of Exponent	Form of Exponent[1]
ES$w.d$	$\lvert exp\rvert \leq 99$	E$\pm z_1z_2$ or $\pm 0z_1z_2$
	$99 < \lvert exp\rvert \leq 999$	$\pm z_1z_2z_3$
ES$w.d$ Ee with $e > 0$	$\lvert exp\rvert \leq 10^e - 1$	E$\pm z_1z_2 \ldots z_e$
ES$w.d$ E0	any	E$\pm z_1z_2 \ldots z_s$
(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 10.13

Examples:

Internal Value	Output field Using SS, ES12.3
`6.421`	`6.421E+00`
`-.5`	`-5.000E-01`
`.00217`	`2.170E-03`
`4721.3`	`4.721E+03`

10.7.2.3.6 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$w.d$Ee 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$w.d$E0, 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 (10.7.2.3.2).

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

$[\pm]\ x_0 \ .\ x_1 x_2 \ldots \mathit{exp}$

where:

- $\pm$ signifies a plus sign or a minus sign;
- . signifies a decimal symbol (10.6);
- $x_0 x_1 x_2 \ldots$ are the most significant hexadecimal digits of the internal value, after rounding if d is not zero (10.7.2.3.8);
- exp is a binary exponent expressed as a decimal integer; for EX$w.d$ and EX$w.d$E0, 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$w.d$Ee 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 10.14

Examples:

Internal value	Edit descriptor	Possible output with SS in effect
1.375	`EX0.1`	`0X1.6P+0`
-15.625	`EX14.4E3`	`-0X1.F400P+003`
1048580.0	`EX0.0`	`0X1.00003P+20`

10.7.2.3.7 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 of the edit descriptors specifies the real part; the second specifies 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.

10.7.2.3.8 Input/output rounding mode

1 The input/output rounding mode can be specified by an OPEN statement (9.5.2), a data transfer statement (9.6.2.13), or an edit descriptor (10.8.7).

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 value closest to the original value and no greater in magnitude than the original value. When the input/output rounding mode is NEAREST, 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 COMPATIBLE, 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 (14.4), NEAREST shall correspond to round to nearest, as specified in ISO/IEC/IEEE 60559:2011.

NOTE 10.15

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.

10.7.2.4 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 R465, R466, R467) 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 10.16

A binary, octal, or hexadecimal constant always consists of at least one digit or hexadecimal digit.

R1022 *hex-digit-string* **is** *hex-digit* [*hex-digit*] ...

6 The output field for the B$w.m$, O$w.m$, and Z$w.m$ edit descriptor is the same as for the Bw, Ow, and Zw edit descriptor, except that the *digit-string* or *hex-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 consists of all zero bits, the output field consists of only blank characters. When m and w are both zero, and the internal value consists of all zero bits, one blank character is produced.

10.7.3 Logical editing

1 The Lw edit descriptor indicates that the field occupies 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 (10.7.5.3).

2 The standard form of the input field consists of optional blanks, optionally followed by a period, followed by a T for true or F for false. The T or 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.

3 A lower-case letter is equivalent to the corresponding upper-case letter in a logical input field.

NOTE 10.17

The logical constants .TRUE. and .FALSE. are acceptable input forms.

4 The output field consists of $w-1$ blanks followed by a T or F, depending on whether the internal value is true or false, respectively.

10.7.4 Character editing

1 The 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 (10.7.5.4). 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.

2 If a field width w is specified with the A edit descriptor, the field consists of w characters. If a field width 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.

3 Let *len* be the length of the input/output list item. If the specified field width w for an A edit descriptor corresponding to an input item is greater than or equal to *len*, the rightmost *len* characters will be taken from the input field. If the specified field width w is less than *len*, the w characters will appear left justified with $len-w$ trailing blanks in the internal value.

4 If the specified field width w for an A edit descriptor corresponding to an output item is greater than *len*, the output field will consist of $w-len$ blanks followed by the *len* characters from the internal value. If the specified field width w is less than or equal to *len*, the output field will consist of the leftmost w characters from the internal value.

NOTE 10.18

For nondefault character kinds, the blank padding character is processor dependent.

5 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 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 10.19

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.

10.7.5 Generalized editing

10.7.5.1 Overview

1 The Gw, G$w.d$ and G$w.d$ 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.

10.7.5.2 Generalized numeric editing

1 When used to specify the input/output of integer, real, and complex data, the Gw, G$w.d$ and G$w.d$ Ee edit descriptors follow the general rules for numeric editing (10.7.2).

NOTE 10.20

The G$w.d$ Ee edit descriptor follows any additional rules for the E$w.d$ Ee edit descriptor.

10.7.5.2.1 Generalized integer editing

1 When used to specify the input/output of integer data, the G$w.d$ and G$w.d$ Ee edit descriptors follow the rules for the Iw edit descriptor (10.7.2.2), 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.

10.7.5.2.2 Generalized real and complex editing

1 The form and interpretation of the input field is the same as for F$w.d$ editing (10.7.2.3.2).

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 G$w.d$Ee 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 G$w.d$ and G$w.d$ Ee edit descriptors is the same as for F$w.d$, 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 $s > d$, kPE$w.d$ or kPE$w.d$Ee editing is used for G$w.d$ editing or G$w.d$Ee editing respectively, where k is the scale factor (10.8.5). If $0 \leq s \leq d$, 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 G$w.d$ editing, $e+2$ for G$w.d$Ee editing if $e > 0$, and 4 for G$w.d$E0 editing.

6 The value of $w-n$ shall be positive.

NOTE 10.21

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.

10.7.5.3 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 (10.7.3). When used to specify the output of logical data, the G0 and G0.d edit descriptors follow the rules for the L1 edit descriptor.

10.7.5.4 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 Aw edit descriptor (10.7.4). 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.

10.7.6 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 (9.6.4.8). The v values of the edit descriptor are passed to the defined input/output procedure as the `v_list` array argument.

NOTE 10.22

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 (9.6.4.8). A DT edit descriptor shall not correspond to a list item that is not of a derived type.

10.8 Control edit descriptors

10.8.1 Position editing

1 The T, TL, TR, and X edit descriptors specify the position at which the next character will be transmitted to or from the record. If any character skipped by a T, TL, TR, or X 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 The position specified by a T edit descriptor may be in either direction from the current position. On input, this allows portions of a record to be processed more than once, possibly with different editing.

3 The position specified by an X edit descriptor is forward from the current position. On input, a position beyond the last character of the record may be specified if no characters are transmitted from such positions.

NOTE 10.23

An nX edit descriptor has the same effect as a TRn edit descriptor.

4 On output, a T, TL, TR, or X 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 T, TL, TR, or X 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 may be replaced. However, a T, TL, TR, or X edit descriptor never directly causes a character already placed in the record to be replaced. Such edit descriptors may result in positioning such that subsequent editing causes a replacement.

10.8.1.1 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 (9.6.4.8.2), 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 Tn edit descriptor indicates that the transmission of the next character to or from a record is to occur at the nth character position of the record, relative to the left tab limit.

3 The TLn 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 TLn 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 TRn 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.

10.8.1.2 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.

10.8.2 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 10.24

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.

10.8.3 Colon editing

1 The colon edit descriptor terminates format control if there are no more effective items in the input/output list (9.6.3). The colon edit descriptor has no effect if there are more effective items in the input/output list.

10.8.4 SS, SP, and S editing

1 The SS, SP, and S edit descriptors temporarily change (9.5.2) the sign mode (9.5.6.17, 9.6.2.14) 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 10.7.2(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.

10.8.5 P editing

1 The kP edit descriptor temporarily changes (9.5.2) the scale factor for the connection to k. The scale factor affects the editing done by the F, E, EN, ES, 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, 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, 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 (R415) 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 and ES editing, the scale factor has no effect.

10.8.6 BN and BZ editing

1 The BN and BZ edit descriptors temporarily change (9.5.2) the blank interpretation mode (9.5.6.6, 9.6.2.6) 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 (10.7.2) and generalized numeric editing (10.7.5.2) on input. It has no effect on output.

10.8.7 RU, RD, RZ, RN, RC, and RP editing

1 The round edit descriptors temporarily change (9.5.2) the connection's input/output rounding mode (9.5.6.16, 9.6.2.13, 10.7.2.3.8). 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.

10.8.8 DC and DP editing

1 The decimal edit descriptors temporarily change (9.5.2) the decimal edit mode (9.5.6.7, 9.6.2.7, 10.6) 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 (10.6) 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.

10.9 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 10.25

A delimiter for a character string edit descriptor is either an apostrophe or quote.

10.10 List-directed formatting

10.10.1 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.

10.10.2 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 10.26

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 10.27

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.

10.10.3 List-directed input

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 10.28

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 Iw 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 (10.7.2.3.2) 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-char*s 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 *digit*s 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.

9 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 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. The effect is as though the sequence were assigned to the next effective item in an intrinsic assignment statement (7.2.1.3).

10.10.3.1 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 10.29

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 10.30

All blanks in a list-directed input record 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 10.31

List-directed input example:

```
INTEGER I; REAL X (8); CHARACTER (11) P;
COMPLEX Z; LOGICAL G
  ...
READ *, I, X, P, Z, G
  ...
```

The input data records are:

NOTE 10.31 (cont.)

```
12345,12345,,2*1.5,4*
ISN'T_BOB'S,(123,0),.TEXAS$
```

The results are:

Variable	Value
I	12345
X (1)	12345.0
X (2)	unchanged
X (3)	1.5
X (4)	1.5
X (5) – X (8)	unchanged
P	ISN'T_BOB'S
Z	(123.0,0.0)
G	true

10.10.4 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 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 (9.6.4.8) 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 Iw 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$w.d$; otherwise, 1PE$w.d$ 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.

8 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.

9 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.

10 Character sequences produced when the delimiter mode has a value of APOSTROPHE are delimited by apostrophes, are preceded and followed by a value separator, and have each internal apostrophe represented on the external medium by two contiguous apostrophes.

11 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 r^*c instead of the sequence of identical values.

12 Slashes, as value separators, and null values are not produced as output by list-directed formatting.

13 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.

NOTE 10.32

The length of the output records is not specified and is processor dependent.

10.11 Namelist formatting

10.11.1 Purpose of namelist formatting

1 Namelist input/output allows data editing with name-value subsequences. This facilitates documentation of input and output files and more flexibility on input.

10.11.2 Name-value subsequences

1 The characters in one or more namelist records constitute a sequence of name-value subsequences, each of which consists of an 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.

2 Each object designator shall begin with a name from the *namelist-group-object-list* (5.8) and shall follow the syntax of *designator* (R601). It shall not contain a vector subscript or an *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 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.

3 A value separator for namelist formatting is the same as for list-directed formatting (10.10.2), or one or more contiguous blanks between a nonblank value and the following object designator or namelist comment (10.11.3.6).

10.11.3 Namelist input

10.11.3.1 Overall syntax

1 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 10.33

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.

2 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.

3 A group name or object name is without regard to case.

10.11.3.2 Namelist input processing

1 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.

2 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 (9.6.3). 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 10.34

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.

3 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.

4 A namelist comment may appear after any value separator except a slash. A namelist comment is also permitted to start in the first nonblank position of an input record except within a character literal constant.

5 Successive namelist records are read by namelist input until a slash is encountered; the remainder of the record is ignored and need not follow the rules for namelist input values.

10.11.3.3 Namelist input values

1 Each value is either a null value (10.11.3.4), 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 (10.11) 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 (10.6). 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 (10.7.2.3.2) 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 (10.7.3).

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 delimited sequence of zero or more *rep-char*s whose kind type parameter is implied by the kind of the corresponding list item. 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 10.35

A character sequence corresponding to a namelist input item of character type shall be delimited either with apostrophes or with quotes. The delimiter is required to avoid ambiguity 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 (9.5.6.8).

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 (7.2.1.3).

10.11.3.4 Null values

1 A null value is specified by

- the $r*$ form,

- blanks between two consecutive nonblank value separators following an equals,
- zero or more blanks preceding the first value separator and 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 10.36

The end of a record following a value separator, with or without intervening blanks, does not specify a null value in namelist input.

10.11.3.5 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.

10.11.3.6 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 10.37

Namelist input example:

```
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:

```
&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:

Variable	Value
I	6
X (1)	12345.0
X (2)	unchanged
X (3)	1.5
X (4)	1.5
X (5) – X (8)	unchanged
P	ISN'T_BOB'S
Z	(123.0,0.0)
G	unchanged

10.11.4 Namelist output

10.11.4.1 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 (9.6.4.8) 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 10.38

The length of the output records is not specified exactly and is processor dependent.

10.11.4.2 Namelist output editing

1 Values in namelist output records are edited as for list-directed output (10.10.4).

NOTE 10.39

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.

10.11.4.3 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 $r*c$ 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* will be produced by namelist formatting at the start of the first output record to indicate which particular group of data objects is being output. A slash is produced by namelist formatting 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.

(Blank page)

11 Program units

11.1 Main program

1 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.

R1101 *main-program* **is** [*program-stmt*]
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-program-stmt

R1102 *program-stmt* **is** PROGRAM *program-name*

R1103 *end-program-stmt* **is** END [PROGRAM [*program-name*]]

C1101 (R1101) 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 11.1

The program name is global to the program (16.2). For explanatory information about uses for the program name, see subclause C.8.1.

NOTE 11.2

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
```

2 The main program may be defined by means other than Fortran; in that case, the program shall not contain a *main-program* program unit.

3 A reference to a Fortran *main-program* shall not appear in any program unit in the program, including itself.

11.2 Modules

11.2.1 Module syntax and semantics

1 A module contains specifications and definitions that are to be accessible to other program units by use association. 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.

2 Procedures and types defined in an intrinsic module are not themselves intrinsic.

R1104 *module* **is** *module-stmt*
[*specification-part*]
[*module-subprogram-part*]
end-module-stmt

R1105 *module-stmt* **is** MODULE *module-name*

R1106 *end-module-stmt* **is** END [MODULE [*module-name*]]

R1107 *module-subprogram-part* **is** *contains-stmt*
[*module-subprogram*] ...

R1108 *module-subprogram* **is** *function-subprogram*
or *subroutine-subprogram*
or *separate-module-subprogram*

C1102 (R1104) If the *module-name* is specified in the *end-module-stmt*, it shall be identical to the *module-name* specified in the *module-stmt*.

C1103 (R1104) A module *specification-part* shall not contain a *stmt-function-stmt*, an *entry-stmt*, or a *format-stmt*.

3 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.

4 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.

NOTE 11.3

The module name is global to the program (16.2).

NOTE 11.4

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.

NOTE 11.5

For a discussion of the impact of modules on dependent compilation, see subclause C.8.2.

NOTE 11.6

For examples of the use of modules, see subclause C.8.3.

11.2.2 The USE statement and use association

1 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.

2 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.

NOTE 11.7

The accessibility of module entities can be controlled by accessibility attributes (4.5.2.2, 5.5.2), and the ONLY option of the USE statement. Definability of module entities can be controlled by the PROTECTED attribute (5.5.15).

R1109	*use-stmt*	**is**	USE [[, *module-nature*] ::] *module-name* [, *rename-list*]
		or	USE [[, *module-nature*] ::] *module-name* , ■ ■ ONLY : [*only-list*]
R1110	*module-nature*	**is**	INTRINSIC
		or	NON_INTRINSIC
R1111	*rename*	**is**	*local-name* => *use-name*
		or	OPERATOR (*local-defined-operator*) => ■ ■ OPERATOR (*use-defined-operator*)
R1112	*only*	**is**	*generic-spec*
		or	*only-use-name*
		or	*rename*
R1113	*only-use-name*	**is**	*use-name*

C1104 (R1109) If *module-nature* is INTRINSIC, *module-name* shall be the name of an intrinsic module.

C1105 (R1109) If *module-nature* is NON_INTRINSIC, *module-name* shall be the name of a nonintrinsic module.

C1106 (R1109) A scoping unit shall not access an intrinsic module and a nonintrinsic module of the same name.

C1107 (R1111) OPERATOR (*use-defined-operator*) shall not identify a type-bound generic interface.

C1108 (R1112) The *generic-spec* shall not identify a type-bound generic interface.

NOTE 11.8

Constraints C1107 and C1108 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*.

C1109 (R1112) Each *generic-spec* shall be a public entity in the module.

C1110 (R1113) Each *use-name* shall be the name of a public entity in the module.

R1114	*local-defined-operator*	**is**	*defined-unary-op*
		or	*defined-binary-op*
R1115	*use-defined-operator*	**is**	*defined-unary-op*
		or	*defined-binary-op*

C1111 (R1115) 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-list*s 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-name*s or *local-defined-operator*s 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 Two or more accessible 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 12.4.3.5. 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 11.9

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 (5.5) 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 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 (5.6.1) of that scoping unit.

NOTE 11.10

The constraints in subclauses 5.9.1, 5.9.2, and 5.8 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 11.11

For a discussion of the impact of the ONLY option and renaming on dependent compilation, see subclause C.8.2.1.

NOTE 11.12

Examples:

```
USE STATS_LIB
```

provides access to all public entities in the module STATS_LIB.

```
USE MATH_LIB; USE STATS_LIB, SPROD => PROD
```

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.

NOTE 11.12 (cont.)

```
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.

11.2.3 Submodules

1 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*.

2 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 11.13

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.

3 A submodule may provide implementations for separate module procedures (12.6.2.5), each of which is declared (12.4.3.2) within that submodule or one of its ancestors, and declarations and definitions of other entities that are accessible by host association in its descendants.

R1116 *submodule* **is** *submodule-stmt*
  [*specification-part*]
  [*module-subprogram-part*]
  end-submodule-stmt

R1117 *submodule-stmt* **is** SUBMODULE (*parent-identifier*) *submodule-name*

R1118 *parent-identifier* **is** *ancestor-module-name* [: *parent-submodule-name*]

R1119 *end-submodule-stmt* **is** END [SUBMODULE [*submodule-name*]]

C1112 (R1116) A submodule *specification-part* shall not contain a *format-stmt*, *entry-stmt*, or *stmt-function-stmt*.

C1113 (R1118) The *ancestor-module-name* shall be the name of a nonintrinsic module; the *parent-submodule-name* shall be the name of a descendant of that module.

C1114 (R1116) If a *submodule-name* appears in the *end-submodule-stmt*, it shall be identical to the one in the *submodule-stmt*.

11.3 Block data program units

1 A block data program unit is used to provide initial values for data objects in named common blocks.

R1120 *block-data* **is** *block-data-stmt*
  [*specification-part*]
  end-block-data-stmt

R1121 *block-data-stmt* **is** BLOCK DATA [*block-data-name*]

R1122 *end-block-data-stmt* **is** END [BLOCK DATA [*block-data-name*]]

C1115 (R1120) 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*.

C1116 (R1120) A *block-data specification-part* shall contain only definitions of derived-type definitions and ASYNCHRONOUS, BIND, COMMON, DATA, DIMENSION, EQUIVALENCE, IMPLICIT, INTRINSIC, PARAMETER, POINTER, SAVE, TARGET, USE, VOLATILE, and type declaration statements.

C1117 (R1120) 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.

12 Procedures

12.1 Concepts

1 The concept of a procedure was introduced in 2.2.3. 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 (12.5.2). A name that appears as a *dummy-arg-name* in the SUBROUTINE, FUNCTION, or ENTRY statement in the declaration of a procedure (R1237) is a dummy argument. Dummy arguments are also specified for intrinsic procedures and procedures in intrinsic modules in Clauses 13, 14, and 15.

12.2 Procedure classifications

12.2.1 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 (7.1.6) within an expression. A reference to a subroutine is a CALL statement, a defined assignment statement (7.2.1.4), the appearance of an object processed by defined input/output (9.6.4.8) in an input/output list, or finalization (4.5.6).

2 A procedure is classified as elemental if it is a procedure that may be referenced elementally (12.8).

12.2.2 Procedure classification by means of definition

12.2.2.1 Intrinsic procedures

1 A procedure that is provided as an inherent part of the processor is an intrinsic procedure.

12.2.2.2 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.

12.2.2.3 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.

12.2.2.4 Procedure pointers

1 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.

12.2.2.5 Statement functions

1 A function that is defined by a single statement is a statement function (12.6.4).

12.3 Characteristics

12.3.1 Characteristics of procedures

1 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.

12.3.2 Characteristics of dummy arguments

12.3.2.1 General

1 Each dummy argument has the characteristic that it is a dummy data object, a dummy procedure, or an asterisk (alternate return indicator).

12.3.2.2 Characteristics of dummy data objects

1 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 (5.5.10, 5.6.9), whether it is optional (5.5.12, 5.6.10), whether it is allocatable (5.5.3), whether it has the ASYNCHRONOUS (5.5.4), CONTIGUOUS (5.5.7), VALUE (5.5.18), or VOLATILE (5.5.19) attributes, whether it is polymorphic, and whether it is a pointer (5.5.14, 5.6.12) or a target (5.5.17, 5.6.15). 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.

12.3.2.3 Characteristics of dummy procedures

1 The characteristics of a dummy procedure are the explicitness of its interface (12.4.2), its characteristics as a procedure if the interface is explicit, whether it is a pointer, and whether it is optional (5.5.12, 5.6.10).

12.3.2.4 Characteristics of asterisk dummy arguments

1 An asterisk as a dummy argument has no characteristics.

12.3.3 Characteristics of function results

1 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. If the length of a character function result is assumed, this is a characteristic.

12.4 Procedure interface

12.4.1 Interface and abstract interface

1 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 (12.6.2.5), the dummy argument names and whether it is recursive shall be the same as in its corresponding module procedure interface body (12.4.3.2).

2 An abstract interface is a set of procedure characteristics with the dummy argument names.

12.4.2 Implicit and explicit interfaces

12.4.2.1 Interfaces and scopes

1 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 12.1

For example, the subroutine LLS of C.8.3.4 has an explicit interface.

12.4.2.2 Explicit interface

1 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 (12.5.2), 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.

12.4.3 Specification of the procedure interface

12.4.3.1 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 12.2

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 (12.4.3.2).

12.4.3.2 Interface block

R1201 *interface-block* **is** *interface-stmt*
    [*interface-specification*] ...
    end-interface-stmt

R1202 *interface-specification* **is** *interface-body*
    or *procedure-stmt*

R1203 *interface-stmt* **is** INTERFACE [*generic-spec*]
    or ABSTRACT INTERFACE

R1204 *end-interface-stmt* **is** END INTERFACE [*generic-spec*]

R1205 *interface-body* **is** *function-stmt*
    [*specification-part*]
    end-function-stmt
    or *subroutine-stmt*
    [*specification-part*]
    end-subroutine-stmt

R1206 *procedure-stmt* **is** [MODULE] PROCEDURE [::] *specific-procedure-list*

R1207 *specific-procedure* **is** *procedure-name*

R1208 *generic-spec* **is** *generic-name*
    or OPERATOR (*defined-operator*)
    or ASSIGNMENT (=)
    or *defined-io-generic-spec*

R1209 *defined-io-generic-spec* **is** READ (FORMATTED)
    or READ (UNFORMATTED)
    or WRITE (FORMATTED)
    or WRITE (UNFORMATTED)

C1201 (R1201) An *interface-block* in a subprogram shall not contain an *interface-body* for a procedure defined by that subprogram.

C1202 (R1201) If the *end-interface-stmt* includes *generic-name*, the *interface-stmt* shall specify the same *generic-name*. If the *end-interface-stmt* includes ASSIGNMENT(=), the *interface-stmt* shall specify ASSIGNMENT(=). If the *end-interface-stmt* includes *defined-io-generic-spec*, the *interface-stmt* shall specify the same *defined-io-generic-spec*. If the *end-interface-stmt* includes OPERATOR(*defined-operator*), the *interface-stmt* shall specify the same *defined-operator*. If one *defined-operator* is .LT., .LE., .GT., .GE., .EQ., or .NE., the other is permitted to be the corresponding operator <, <=, >, >=, ==, or /=.

C1203 (R1203) If the *interface-stmt* is ABSTRACT INTERFACE, then the *function-name* in the *function-stmt* or the *subroutine-name* in the *subroutine-stmt* shall not be the same as a keyword that specifies an intrinsic type.

C1204 (R1202) A *procedure-stmt* is allowed only in an interface block that has a *generic-spec*.

C1205 (R1205) 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.

C1206 (R1205) An *interface-body* shall not contain a *data-stmt*, *format-stmt*, *entry-stmt*, or *stmt-function-stmt*.

C1207 (R1206) If MODULE appears in a *procedure-stmt*, each *procedure-name* in that statement shall denote a module procedure.

C1208 (R1207) A *procedure-name* shall denote a nonintrinsic procedure that has an explicit interface.

C1209 (R1201) 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 the *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 (12.6.2.5). 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 (12.6.2.5) 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 or a dummy procedure. 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; otherwise, it is an external procedure.

6 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 (12.4.3.7), 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 12.3

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 12.4

An example of a specific interface block is:

```
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 (12.5.2); for example:

```
PRINT *, EXT3 (Q = P_MASK (N+1 : N+1000), P = ACTUAL_P)
```

12.4.3.3 GENERIC statement

1 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.

R1210	*generic-stmt*	**is**	GENERIC [, *access-spec*] :: *generic-spec* => *specific-procedure-list*

2 If *access-spec* appears, it specifies the accessibility (5.5.2) of *generic-spec*.

12.4.3.4 IMPORT statement

R1211	*import-stmt*	**is**	IMPORT [[::] *import-name-list*]
		or	IMPORT, ONLY : *import-name-list*
		or	IMPORT, NONE
		or	IMPORT, ALL

C1210 (R1211) An IMPORT statement shall not appear in the *specification-part* of a *main-program*, *external-subprogram*, *module*, or *block-data*.

C1211 (R1211) Each *import-name* shall be the name of an entity in the host scoping unit.

C1212 If any IMPORT statement in a scoping unit has an ONLY specifier, all IMPORT statements in that scoping unit shall have an ONLY specifier.

C1213 IMPORT, NONE shall not appear in a submodule.

C1214 If an IMPORT, NONE or IMPORT, ALL statement appears in a scoping unit, no other IMPORT statement shall appear in that scoping unit.

C1215 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.

C1216 An entity whose name appears as an *import-name* or which is made accessible by IMPORT ALL shall not appear in any context described in 16.5.1.4 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 16.5.1.4 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 12.5

IMPORT NONE can be used to prevent accidental host association:

```
SUBROUTINE s(x,n)
  IMPLICIT NONE
  IMPORT NONE
  ...
  DO i=1,n ! Forces I to be locally declared.
```

NOTE 12.6

IMPORT ALL 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 must be the host X.
```

NOTE 12.7

IMPORT ONLY can be used to document deliberate access via host association whilst blocking accidental access:

```
SUBROUTINE sub
  IMPORT,ONLY :: x, y
  ...
  x = y + z ! X and Y imported, Z must be local.
```

NOTE 12.8

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
```

NOTE 12.8 (cont.)

```
    ...
  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)
        IMPORT T
        INTEGER,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.

12.4.3.5 Generic interfaces

12.4.3.5.1 Generic identifiers

1 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.

2 The *generic-spec* in an *interface-stmt* is a generic identifier for all the procedures in the interface block. The rules specifying how any two procedures with the same generic identifier shall differ are given in 12.4.3.5.5. 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. The *generic-spec* in a GENERIC statement is a generic identifier for all of the procedures named in its *specific-procedure-list*.

3 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.

4 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.

5 An *interface-stmt* having a *defined-io-generic-spec* is an interface for a defined input/output procedure (9.6.4.8).

NOTE 12.9

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
```

NOTE 12.9 (cont.)

```
   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 12.10

A *type-bound-generic-stmt* within a derived-type definition (4.5.5) specifies a generic identifier for a set of type-bound procedures.

12.4.3.5.2 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 (7.1.6, 12.5). 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* (R308), 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 (7.1.5).

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 (7.1.6.2), extending one form (such as <=) has the effect of defining both forms (<= and .LE.).

NOTE 12.11

An example of the use of the OPERATOR generic specification is:

```
INTERFACE OPERATOR ( * )
   FUNCTION BOOLEAN_AND (B1, B2)
      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

NOTE 12.11 (cont.)

```
BOOLEAN_AND (SENSOR (1:N), ACTION (1:N))    ! SENSOR and ACTION are
                                            ! of type LOGICAL
```

12.4.3.5.3 Defined assignments

1 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 (7.2.1.4, 7.2.1.5). 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.

2 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 7.8, 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 12.12

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 12.13

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.

12.4.3.5.4 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 9.6.4.8.3.

12.4.3.5.5 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 12.14

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.

C1217 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.

C1218 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.

C1219 Within the scope of a *defined-io-generic-spec*, if two procedures have that generic identifier, their `dtv` arguments (9.6.4.8.3) shall be distinguishable.

C1220 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 12.15

An extensive explanation of the application of these rules is in C.9.6.

12.4.3.6 EXTERNAL statement

1 An EXTERNAL statement specifies the EXTERNAL attribute (5.5.9) for a list of names.

R1212 *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 12.16

For explanatory information on potential portability problems with external procedures, see subclause C.9.1.

NOTE 12.17

An example of an EXTERNAL statement is:

```
EXTERNAL FOCUS
```

12.4.3.7 Procedure declaration statement

1 A procedure declaration statement declares procedure pointers, dummy procedures, and external procedures. It specifies the EXTERNAL attribute (5.5.9) for all entities in the *proc-decl-list*.

R1213	*procedure-declaration-stmt*	**is**	PROCEDURE ([*proc-interface*]) ■
			■ [[, *proc-attr-spec*] ... ::] *proc-decl-list*
R1214	*proc-interface*	**is**	*interface-name*
		or	*declaration-type-spec*
R1215	*proc-attr-spec*	**is**	*access-spec*
		or	*proc-language-binding-spec*
		or	INTENT (*intent-spec*)
		or	OPTIONAL
		or	POINTER
		or	PROTECTED
		or	SAVE
R1216	*proc-decl*	**is**	*procedure-entity-name* [=> *proc-pointer-init*]
R1217	*interface-name*	**is**	*name*
R1218	*proc-pointer-init*	**is**	*null-init*
		or	*initial-proc-target*

R1219 *initial-proc-target* **is** *procedure-name*

C1221 (R1217) 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 13.2.

C1222 (R1217) The *name* shall not be the same as a keyword that specifies an intrinsic type.

C1223 (R1213) If a *proc-interface* describes an elemental procedure, each *procedure-entity-name* shall specify an external procedure.

C1224 (R1216) If => appears in *proc-decl*, the procedure entity shall have the POINTER attribute.

C1225 (R1219) The *procedure-name* shall be the name of a nonelemental external or module procedure, or a specific intrinsic function listed in Table 13.2.

C1226 (R1213) If *proc-language-binding-spec* with a 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.

C1227 (R1213) 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 (12.4.3.2) for the declared procedure entities. The abstract interface (12.4) 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 (12.6.2.2) 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 5.5. If a *proc-language-binding-spec* with NAME= appears, it specifies a binding label or its absence, as described in 15.10.2. A *proc-language-binding-spec* without a NAME= is allowed, but is redundant with the *proc-interface* required by C1227.

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.

7 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.

8 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.

9 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.

10 If *initial-proc-target* and *procedure-entity-name* are functions, their results shall have the same characteristics.

NOTE 12.18

The following code illustrates procedure declaration statements. Note 7.46 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
```

12.4.3.8 INTRINSIC statement

1 An INTRINSIC statement specifies the INTRINSIC attribute (5.5.11) for a list of names.

R1220 *intrinsic-stmt* **is** INTRINSIC [::] *intrinsic-procedure-name-list*

C1228 (R1220) Each *intrinsic-procedure-name* shall be the name of an intrinsic procedure.

12.4.3.9 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 dummy arguments of a procedure invoked from where the interface of the procedure is implicit shall be such that the actual arguments are consistent with the characteristics of the dummy arguments.

12.5 Procedure reference

12.5.1 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.

R1221 *function-reference* **is** *procedure-designator* ([*actual-arg-spec-list*])

C1229 (R1221) The *procedure-designator* shall designate a function.

C1230 (R1221) The *actual-arg-spec-list* shall not contain an *alt-return-spec*.

R1222 *call-stmt* **is** CALL *procedure-designator* [([*actual-arg-spec-list*])]

C1231 (R1222) The *procedure-designator* shall designate a subroutine.

R1223 *procedure-designator* **is** *procedure-name*
or *proc-component-ref*
or *data-ref* % *binding-name*

C1232 (R1223) A *procedure-name* shall be a generic name or the name of a procedure.

C1233 (R1223) A *binding-name* shall be a binding name (4.5.5) of the declared type of *data-ref*.

C1234 (R1223) A *data-ref* shall not be a polymorphic subobject of a coindexed object.

C1235 (R1223) 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 12.5.6.

4 A function may also be referenced as a defined operation (7.1.6). A subroutine may also be referenced as a defined assignment (7.2.1.4, 7.2.1.5), by defined input/output (9.6.4.8), or by finalization (4.5.6).

NOTE 12.19

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.

R1224 *actual-arg-spec* **is** [*keyword* =] *actual-arg*

R1225 *actual-arg* **is** *expr*
or *variable*
or *procedure-name*
or *proc-component-ref*
or *alt-return-spec*

R1226 *alt-return-spec* **is** * *label*

C1236 (R1224) The *keyword* = shall not appear if the interface of the procedure is implicit.

C1237 (R1224) 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.

C1238 (R1224) Each *keyword* shall be the name of a dummy argument in the explicit interface of the procedure.

C1239 (R1225) A nonintrinsic elemental procedure shall not be used as an actual argument.

C1240 (R1225) A *procedure-name* shall be the name of an external, internal, module, or dummy procedure, a specific intrinsic function listed in Table 13.2, or a procedure pointer.

C1241 (R1225) *expr* shall not be a variable.

C1242 (R1226) The *label* shall be the statement label of a branch target statement that appears in the same inclusive scope as the *call-stmt*.

C1243 An actual argument that is a coindexed object shall not have a pointer ultimate component.

NOTE 12.20

Examples of procedure reference using procedure pointers:

```
P => BESSEL
WRITE (*, *) P(2.5)     !-- BESSEL(2.5)

S => PRINT_REAL
CALL S(3.14)
```

NOTE 12.21

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:

```
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 must be explicit and available.

CONTAINS

   REAL(C_FLOAT) FUNCTION MY_F(X) BIND(C) ! Integrand
      REAL(C_FLOAT), VALUE :: X
```

NOTE 12.21 (cont.)

```
      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.

12.5.2 Actual arguments, dummy arguments, and argument association

12.5.2.1 Argument correspondence

1 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 (4.5.4.5), 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 12.22

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
   ...
```

can be invoked with

```
CALL SOLVE (FUN, SOL, PRINT = 6)
```

provided its interface is explicit, and if the interface is specified by an interface block, the name of the last argument is PRINT.

12.5.2.2 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 (4.5.4.5), the *data-ref* of the *function-reference* or *call-stmt* corresponds, as an actual argument, with the passed-object dummy argument.

12.5.2.3 Argument association

1 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.

2 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.

3 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.

4 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.

5 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.

6 The entity that is argument associated with a dummy argument is called its effective argument.

7 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 12.23

For the sequence of subroutine calls

```
INTEGER :: X(100)
CALL SUBA (X)
...
SUBROUTINE SUBA(A)
INTEGER :: A(:)
CALL SUBB (A(1:5), A(5:1:-1))
...
SUBROUTINE SUBB(B, C)
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 12.24

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.

12.5.2.4 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 (15.2.2) 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 an 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 (12.5.2.11).

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 12.25

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 (16.5.2.5) 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 (15.2.2), 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 (12.8), each element of an array actual argument or of a sequence in a sequence association (12.5.2.11) is associated with the element of the dummy array that has the same position in array element order (6.5.3.2).

NOTE 12.26

For default character sequence associations, the interpretation of element is provided in 12.5.2.11.

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.

19 If the procedure is nonelemental 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.

NOTE 12.27

Argument intent specifications serve several purposes. See Note 5.17.

NOTE 12.28

For more explanatory information on targets as dummy arguments, see subclause C.9.4.

C1244 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.

C1245 (R1225) If an actual argument is a nonpointer array that has the ASYNCHRONOUS or VOLATILE attribute but is not simply contiguous (6.5.4), and the corresponding dummy argument has either the VOLATILE or ASYNCHRONOUS attribute, that dummy argument shall be assumed-shape or assumed-rank and shall not have the CONTIGUOUS attribute.

C1246 (R1225) 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 VOLATILE or ASYNCHRONOUS 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 12.29

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.

12.5.2.5 Allocatable and pointer dummy variables

1 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.

2 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 12.30

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.

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. The type parameter values of the actual argument shall agree with the corresponding ones of the dummy argument that are not assumed or deferred.

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.

12.5.2.6 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 The values of assumed type parameters of a dummy argument are assumed from the corresponding type parameters of its effective argument.

5 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.

6 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 (16.5.2.5) and are associated with the dummy argument remain associated with the actual argument.

7 If a dummy argument has INTENT (OUT) or INTENT (INOUT), the actual argument shall be definable. If a dummy argument has INTENT (OUT), an allocated actual argument is deallocated on procedure invocation (6.7.3.2).

12.5.2.7 Pointer dummy variables

1 The requirements in this subclause apply to actual arguments that correspond to dummy data pointers.

C1247 The actual argument corresponding to a dummy pointer with the CONTIGUOUS attribute shall be simply contiguous (6.5.4).

C1248 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 The nondeferred type parameters and ranks shall agree. The values of assumed type parameters of a dummy argument are assumed from the corresponding type parameters of its effective argument.

4 If the dummy argument has INTENT (OUT), the pointer association status of the actual argument becomes undefined on invocation of the procedure.

5 If the dummy argument is nonoptional and the actual argument is allocatable, the actual argument shall be allocated.

NOTE 12.31

For more explanatory information on pointers as dummy arguments, see subclause C.9.4.

12.5.2.8 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 12.32

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 12.33

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 12.34

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.

12.5.2.9 Actual arguments associated with dummy procedure entities

1 If the interface of a dummy procedure is explicit, its characteristics as a procedure (12.3.1) 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 13.2. 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 (12.6.2.4).

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.

12.5.2.10 Actual arguments and alternate return indicators

1 If a dummy argument is an asterisk (12.6.2.3), the corresponding actual argument shall be an alternate return specifier (R1226).

12.5.2.11 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 (15.2.2). 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 12.35

Some of the elements in the element sequence might consist of storage units from different elements of the original array.

4 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.

12.5.2.12 Argument presence and restrictions on arguments not present

1 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.

2 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.

3 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 a SELECT TYPE or ASSOCIATE construct.

(11) It shall not be supplied as the *data-ref* in a *procedure-designator*.

(12) If shall not be supplied as the *scalar-variable* in a *proc-component-ref*.

4 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.

12.5.2.13 Restrictions on entities associated with dummy arguments

1 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 an array section with a vector subscript, or

(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.

(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 an array section with a vector subscript, or

(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.

NOTE 12.36

In

```
SUBROUTINE OUTER
   REAL, POINTER :: A (:)
   ...
   ALLOCATE (A (1:N))
   ...
   CALL INNER (A)
   ...
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

```
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,

NOTE 12.36 (cont.)

```
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 12.37

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 shall not be defined, redefined, or become undefined during the execution of the procedure. For example, in

```
CALL SUB (A (1:5), A (3:9))
```

A (3:5) shall not 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 shall not be defined, redefined, or become undefined through the second dummy argument because it is part of the argument associated with the first dummy argument. A (1:2) remains definable through the first dummy argument and A (6:9) remains definable through the second dummy argument.

NOTE 12.38

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.
```

B (3:5) cannot be defined because it is part of the argument associated with the second dummy argument. C (1:3) cannot be defined because it is part of the argument associated with the first dummy argument. A (1:2) [which is B (1:2)] remains definable through the first dummy argument and A (6:9) [which is C (4:7)] remains definable through the second dummy argument.

NOTE 12.39

In

```
MODULE DATA
   REAL :: W, X, Y, Z
END MODULE DATA

PROGRAM MAIN
   USE DATA
      ...
   CALL INIT (X)
      ...
END PROGRAM MAIN
```

NOTE 12.39 (cont.)

```
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 12.40

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.

12.5.3 Function reference

1 A function is invoked during expression evaluation by a *function-reference* or by a defined operation (7.1.6). 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 (12.3.3) are determined by the interface of the function. If a reference to an elemental function (12.8) is an elemental reference, all array arguments shall have the same shape.

12.5.4 Subroutine reference

1 A subroutine is invoked by execution of a CALL statement, execution of a defined assignment statement (7.2.1.4), defined input/output (9.6.4.8.2), or finalization(4.5.6). 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 (12.8) 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.

12.5.5 Resolving named procedure references

12.5.5.1 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 an INTRINSIC attribute specification for that name and it is the generic name of an intrinsic procedure;
(3) 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

(4) 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.

12.5.5.2 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 12.4.3.5.5 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 12.4.3.5.5 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.

5 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 12.41

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_RANDOM(X)
     REAL X(:)
```

NOTE 12.41 (cont.)

```
     REAL VECTOR_RANDOM(SIZE(X))
     END FUNCTION VECTOR_RANDOM
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_RANDOM.

NOTE 12.42

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.

12.5.5.3 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 12.43

Because of the renaming facility of the USE statement, the name in the reference can be different from the original name of the procedure.

12.5.5.4 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.

12.5.6 Resolving type-bound procedure references

1 If the *binding-name* in a *procedure-designator* (R1223) 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*.

12.6 Procedure definition

12.6.1 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 13, but may include others. However, a standard-conforming program shall not make use of intrinsic procedures other than those described in Clause 13.

12.6.2 Procedures defined by subprograms

12.6.2.1 General

1 A subprogram defines one or more procedures. A procedure is defined by the initial SUBROUTINE or FUNCTION statement, and each ENTRY statement defines an additional procedure (12.6.2.6).

2 A subprogram is specified to be elemental (12.8), pure (12.7), recursive, or a separate module subprogram (12.6.2.5) by a *prefix-spec* in its initial SUBROUTINE or FUNCTION statement.

R1227	*prefix*	**is**	*prefix-spec* [*prefix-spec*] ...
R1228	*prefix-spec*	**is**	*declaration-type-spec*
		or	ELEMENTAL
		or	IMPURE
		or	MODULE
		or	NON_RECURSIVE
		or	PURE
		or	RECURSIVE

C1249 (R1227) A *prefix* shall contain at most one of each *prefix-spec*.

C1250 (R1227) A *prefix* shall not specify both PURE and IMPURE.

C1251 (R1227) A *prefix* shall not specify both NON_RECURSIVE and RECURSIVE.

C1252 An elemental procedure shall not have the BIND attribute.

C1253 (R1227) 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.

C1254 (R1227) 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.

C1255 (R1227) 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.

C1256 (R1227) 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.

C1257 (R1227) 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.

3 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.

4 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 12.7.

5 If the *prefix-spec* ELEMENTAL appears, the subprogram is an elemental subprogram and shall meet the additional constraints of 12.8.1.

12.6.2.2 Function subprogram

1 A function subprogram is a subprogram that has a FUNCTION statement as its first statement.

R1229 *function-subprogram* **is** *function-stmt*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-function-stmt

R1230 *function-stmt* **is** [*prefix*] FUNCTION *function-name* ■
■ ([*dummy-arg-name-list*]) [*suffix*]

C1258 (R1230) 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.

C1259 (R1230) If RESULT appears, the *function-name* shall not appear in any specification statement in the scoping unit of the function subprogram.

R1231 *proc-language-binding-spec* **is** *language-binding-spec*

C1260 (R1231) 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.

C1261 If *proc-language-binding-spec* is specified for a procedure, each of its dummy arguments shall be an interoperable procedure (15.3.7) or a variable that is interoperable (15.3.5, 15.3.6), assumed-shape, assumed-rank, assumed-type, of type CHARACTER with assumed length, or that has the ALLOCATABLE or POINTER attributes.

C1262 If *proc-language-binding-spec* is specified for a function, the function result shall be an interoperable scalar variable.

C1263 A variable that is a dummy argument of a procedure that has a *proc-language-binding-spec* shall not have both the OPTIONAL and VALUE attributes.

C1264 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.

R1232 *dummy-arg-name* **is** *name*

C1265 (R1232) A *dummy-arg-name* shall be the name of a dummy argument.

R1233 *suffix* **is** *proc-language-binding-spec* [RESULT (*result-name*)]
or RESULT (*result-name*) [*proc-language-binding-spec*]

R1234 *end-function-stmt* **is** END [FUNCTION [*function-name*]]

C1266 (R1229) An internal function subprogram shall not contain an *internal-subprogram-part*.

C1267 (R1234) 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 (12.3.1).

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 12.44

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 12.45

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

```
  int FrEd(int n, int m, float x);
```

12.6.2.3 Subroutine subprogram

1 A subroutine subprogram is a subprogram that has a SUBROUTINE statement as its first statement.

R1235 *subroutine-subprogram* **is** *subroutine-stmt*
  [*specification-part*]
  [*execution-part*]
  [*internal-subprogram-part*]
  end-subroutine-stmt

R1236 *subroutine-stmt* **is** [*prefix*] SUBROUTINE *subroutine-name* ■
  ■ [([*dummy-arg-list*]) [*proc-language-binding-spec*]]

C1268 (R1236) The *prefix* of a *subroutine-stmt* shall not contain a *declaration-type-spec*.

R1237 *dummy-arg* **is** *dummy-arg-name*
  or *

R1238 *end-subroutine-stmt* **is** END [SUBROUTINE [*subroutine-name*]]

C1269 (R1235) An internal subroutine subprogram shall not contain an *internal-subprogram-part*.

C1270 (R1238) If a *subroutine-name* appears in the *end-subroutine-stmt*, it shall be identical to the *subroutine-name* specified in the *subroutine-stmt*.

2 The name of the subroutine is *subroutine-name*.

12.6.2.4 Instances of a subprogram

1 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 procedure pointers. Saved local entities are shared by all instances of the subprogram.

2 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 (16.5.5). 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.

12.6.2.5 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.

R1239 *separate-module-subprogram* **is** *mp-subprogram-stmt*
  [*specification-part*]
  [*execution-part*]
  [*internal-subprogram-part*]
  end-mp-subprogram-stmt

R1240 *mp-subprogram-stmt* **is** MODULE PROCEDURE *procedure-name*

R1241 *end-mp-subprogram-stmt* **is** END [PROCEDURE [*procedure-name*]]

C1271 (R1239) 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.

C1272 (R1241) 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 12.46

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.

12.6.2.6 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.

R1242 *entry-stmt* **is** ENTRY *entry-name* [([*dummy-arg-list*]) [*suffix*]]

C1273 (R1242) If RESULT appears, the *entry-name* shall not appear in any specification or type declaration statement in the scoping unit of the function program.

C1274 (R1242) 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*.

C1275 (R1242) RESULT shall appear only if the *entry-stmt* is in a function subprogram.

C1276 (R1242) A *dummy-arg* shall not be an alternate return indicator if the ENTRY statement is in a function subprogram.

C1277 (R1242) 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.

2 Optionally, a subprogram may have one or more ENTRY statements.

3 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.

4 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.

5 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.

6 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 (12.5).

7 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.

8 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.

9 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.

10 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 (12.5.2.12).

11 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.

12 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.

13 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.

12.6.2.7 RETURN statement

R1243 *return-stmt* **is** RETURN [*scalar-int-expr*]

C1278 (R1243) The *return-stmt* shall be in the inclusive scope of a function or subroutine subprogram.

C1279 (R1243) The *scalar-int-expr* is allowed only in the inclusive scope of a subroutine subprogram.

1 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 (8.2) 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 *end-function-stmt*, *end-mp-subprogram-stmt*, or *end-subroutine-stmt* is equivalent to execution of a RETURN statement with no expression.

12.6.2.8 CONTAINS statement

R1244 *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 may contain, or it introduces the type-bound procedure part of a derived-type definition (4.5.5). The CONTAINS statement is not executable.

12.6.3 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 12.47

Examples of code that might cause a transfer of control that bypasses the normal return mechanism of a Fortran procedure are setjmp and longjmp in C and exception handling in other languages. No such behavior is permitted by this part of ISO/IEC 1539.

3 If the interface of a procedure has a *proc-language-binding-spec*, the procedure is interoperable (15.10).

4 Interoperation with C functions is described in 15.10.

NOTE 12.48

For explanatory information on definition of procedures by means other than Fortran, see subclause C.9.2.

12.6.4 Statement function

1 A statement function is a function defined by a single statement.

R1245 *stmt-function-stmt* **is** *function-name* ([*dummy-arg-name-list*]) = *scalar-expr*

C1280 (R1245) Each *primary* in *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 *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 *scalar-expr*, its definition shall have been provided earlier in the scoping unit and shall not be the name of the statement function being defined.

C1281 (R1245) Named constants in *scalar-expr* shall have been declared earlier in the scoping unit or made accessible by use or host association. If array elements appear in *scalar-expr*, the array shall have been declared as an array earlier in the scoping unit or made accessible by use or host association.

C1282 (R1245) If a *dummy-arg-name*, variable, function reference, or dummy function reference is typed by the implicit typing rules, its appearance in any subsequent type declaration statement shall confirm this implied type and the values of any implied type parameters.

C1283 (R1245) The *function-name* and each *dummy-arg-name* shall be specified, explicitly or implicitly, to be scalar.

C1284 (R1245) A given *dummy-arg-name* shall not appear more than once in any *dummy-arg-name-list*.

2 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.

3 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.

4 A statement function shall not be supplied as an actual argument.

5 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.

6 A function reference in the scalar expression shall not cause a dummy argument of the statement function to become redefined or undefined.

12.7 Pure procedures

1 A pure procedure is

- a pure intrinsic procedure (13.1),
- 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.

2 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.

C1285 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.

C1286 The function result of a pure function shall not be such that finalization of a reference to the function would reference an impure procedure.

C1287 A pure function shall not have a polymorphic allocatable result.

C1288 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.

C1289 An INTENT (OUT) dummy argument of a pure procedure shall not be such that finalization of the actual argument would reference an impure procedure.

C1290 An INTENT (OUT) dummy argument of a pure procedure shall not be polymorphic.

C1291 A local variable of a pure subprogram, or of a BLOCK construct within a pure subprogram, shall not have the SAVE attribute.

NOTE 12.49

Variable initialization in a *type-declaration-stmt* or a *data-stmt* implies the SAVE attribute; therefore, such initialization is also disallowed.

C1292 The *specification-part* of a pure subprogram shall specify that all its dummy procedures are pure.

C1293 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.

C1294 All internal subprograms in a pure subprogram shall be pure.

C1295 A *designator* of a variable with the VOLATILE attribute shall not appear in a pure subprogram.

C1296 In a pure subprogram any designator with a base object that is in common or accessed by host or use association, is a 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 (16.6.7),
(2) in a pointer association context (16.6.8),
(3) as the *data-target* in a *pointer-assignment-stmt*,
(4) as the *expr* corresponding to a component with the POINTER attribute in a *structure-constructor*,
(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, or
(7) as an actual argument corresponding to a dummy argument with the POINTER attribute.

NOTE 12.50

Item 5 requires that processors be able to determine if entities with the PRIVATE attribute or with private components have a pointer component.

C1297 Any procedure referenced in a pure subprogram, including one referenced via a defined operation, defined assignment, defined input/output, or finalization, shall be pure.

C1298 A statement that might result in the deallocation of a polymorphic entity is not permitted in a pure procedure.

NOTE 12.51

Apart from the DEALLOCATE statement, this includes intrinsic assignment if the variable has a potential subobject component that is polymorphic and allocatable.

C1299 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*.

C12100 A pure subprogram shall not contain a *read-stmt* or *write-stmt* whose *io-unit* is a *file-unit-number* or *.

C12101 A pure subprogram shall not contain an image control statement (8.5.1).

NOTE 12.52

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 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 12.53

Pure subroutines are included to allow subroutine calls from pure procedures in a safe way, and to allow *forall-assignment-stmt*s 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.

12.8 Elemental procedures

12.8.1 Elemental procedure declaration and interface

1 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.

2 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.

C12102 All dummy arguments of an elemental procedure shall be scalar noncoarray dummy data objects and shall not have the POINTER or ALLOCATABLE attribute.

C12103 The result of an elemental function shall be scalar, and shall not have the POINTER or ALLOCATABLE attribute.

C12104 The *specification-part* of an elemental subprogram shall specify the intents of all of its dummy arguments that do not have the VALUE attribute.

C12105 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 (7.1.11), and that specification inquiry shall not depend on a property that is deferred.

3 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.

12.8.2 Elemental function actual arguments and results

1 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 12.54

An example of an elemental reference to the intrinsic function MAX:

if X and Y are arrays of shape (M, N),

```
MAX (X, 0.0, Y)
```

is an array expression of shape (M, N) whose elements have values

```
MAX (X(I, J),  0.0, Y(I, J)), I = 1, 2, ..., M, J = 1,2, ..., N
```

12.8.3 Elemental subroutine actual arguments

1 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.

13 Intrinsic procedures and modules

13.1 Classes of intrinsic procedures

1 Intrinsic procedures are divided into seven classes: inquiry functions, elemental functions, transformational functions, elemental subroutines, pure subroutines, atomic subroutines, and (impure) subroutines.

2 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 12.8. 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.

3 An atomic subroutine is an intrinsic subroutine that performs an action on its ATOM argument atomically. The effect of executing an atomic subroutine is as if the subroutine were executed instantaneously, thus not overlapping other atomic actions that might occur asynchronously. The sequence of atomic actions within ordered segments is specified in 2.3.5. How sequences of atomic actions in unordered segments interleave with each other is processor dependent, and the order of accesses to affected variables may appear to be inconsistent between different images or between different variables.

NOTE 13.1

The most reliable way to use atomic subroutines is for a single image to define a particular variable, repeatedly, and for another image to inspect its changes. However, even this use is processor dependent.

4 The subroutine MOVE_ALLOC and the elemental subroutine MVBITS are pure. No other standard intrinsic subroutine is pure.

5 Generic names of standard intrinsic procedures are listed in 13.5. 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 13.6.

6 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 (12.5.2.9).

7 Elemental intrinsic procedures behave as described in 12.8.

13.2 Arguments to intrinsic procedures

13.2.1 General rules

1 All intrinsic procedures may be invoked with either positional arguments or argument keywords (12.5). The descriptions in 13.5 through 13.7 give the argument keyword names and positional sequence for standard intrinsic procedures.

2 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 12.5.2.

NOTE 13.2

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 13.3

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.

3 The dummy arguments of the specific intrinsic procedures in 13.6 have INTENT (IN). The dummy arguments of the intrinsic procedures in 13.7 have INTENT (IN) if the intent is not stated explicitly.

4 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.

5 Intrinsic subroutines that assign values to arguments of type character do so in accordance with the rules of intrinsic assignment (7.2.1.3).

6 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 12.5.2.13.

7 An argument to an intrinsic procedure other than ASSOCIATED, NULL, or PRESENT shall be a data object.

13.2.2 The shape of array arguments

1 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.

13.2.3 Mask arguments

1 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.

2 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.

13.2.4 DIM arguments and reduction functions

1 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.

2 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.

13.3 Bit model

13.3.1 General

1 The bit manipulation procedures are described in terms of a model for the representation and behavior of bits on a processor.

2 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 $w_{z-1} \dots w_0$, 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 13.4 would have $z = 32$, thereby defining a 32-bit integer.

3 The interpretation of a negative integer as a sequence of bits is processor dependent.

4 The inquiry function BIT_SIZE provides the value of the parameter z of the model.

5 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 13.4 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.

13.3.2 Bit sequence comparisons

1 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.

2 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.

13.3.3 Bit sequences as arguments to INT and REAL

1 When a *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 *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 *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.

C1301 If a *boz-literal-constant* is truncated as an argument to the intrinsic function REAL, the discarded bits shall all be zero.

NOTE 13.4

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 *boz-literal-constant* arguments to those intrinsic functions is the same as for the intrinsic function REAL.

13.4 Numeric models

1 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.

2 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.

3 The model set for real x is defined by

$$x = \begin{cases} 0 \text{ or} \\ s \times b^e \times \displaystyle\sum_{k=1}^{p} f_k \times b^{-k} , \end{cases}$$

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 $e_{\max}$ and some integer minimum $e_{\min}$ 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, $e_{\min}$, and $e_{\max}$ determine the set of model floating-point numbers.

4 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.

5 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 13.5

Examples of these functions in 13.7 use the models

$$i = s \times \sum_{k=0}^{30} w_k \times 2^k$$

and

$$x = 0 \text{ or } s \times 2^e \times \left(\frac{1}{2} + \sum_{k=2}^{24} f_k \times 2^{-k} \right), \quad -126 \le e \le 127$$

13.5 Standard generic intrinsic procedures

1 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 13.6

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 13.7

Many of the argument keywords have names that are indicative of their usage. For example:

KIND	Describes the kind type parameter of the result
STRING, STRING_A	An arbitrary character string
BACK	Controls the direction of string scan (forward or backward)
MASK	A mask to be applied to the arguments
DIM	A selected dimension of an array argument

2 In the Class column of Table 13.1,

- A indicates that the procedure is an atomic 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 13.1: **Standard generic intrinsic procedure summary**

Procedure	Arguments	Class	Description
ABS	(A)	E	Absolute value.
ACHAR	(I [, KIND])	E	Character from ASCII code value.
ACOS	(X)	E	Arccosine (inverse cosine) function.
ACOSH	(X)	E	Inverse hyperbolic cosine function.
ADJUSTL	(STRING)	E	Left-justified string value.
ADJUSTR	(STRING)	E	Right-justified string value.
AIMAG	(Z)	E	Imaginary part of a complex number.
AINT	(A [, KIND])	E	Truncation toward 0 to a whole number.
ALL	(MASK) or (MASK, DIM)	T	Array reduced by .AND. operator.
ALLOCATED	(ARRAY) or (SCALAR)	I	Allocation status of allocatable variable.
ANINT	(A [, KIND])	E	Nearest whole number.
ANY	(MASK) or (MASK, DIM)	T	Array reduced by .OR. operator.
ASIN	(X)	E	Arcsine (inverse sine) function.
ASINH	(X)	E	Inverse hyperbolic sine function.
ASSOCIATED	(POINTER [, TARGET])	I	Pointer association status inquiry.
ATAN	(X) or (Y, X)	E	Arctangent (inverse tangent) function.
ATAN2	(Y, X)	E	Arctangent (inverse tangent) function.
ATANH	(X)	E	Inverse hyperbolic tangent function.
ATOMIC_DEFINE	(ATOM, VALUE)	A	Define a variable atomically.
ATOMIC_REF	(VALUE, ATOM)	A	Reference a variable atomically.
BESSEL_J0	(X)	E	Bessel function of the 1^{st} kind, order 0.
BESSEL_J1	(X)	E	Bessel function of the 1^{st} kind, order 1.
BESSEL_JN	(N, X)	E	Bessel function of the 1^{st} kind, order N.
BESSEL_JN	(N1, N2, X)	T	Bessel functions of the 1^{st} kind.
BESSEL_Y0	(X)	E	Bessel function of the 2^{nd} kind, order 0.
BESSEL_Y1	(X)	E	Bessel function of the 2^{nd} kind, order 1.
BESSEL_YN	(N, X)	E	Bessel function of the 2^{nd} kind, order N.
BESSEL_YN	(N1, N2, X)	T	Bessel functions of the 2^{nd} kind.
BGE	(I, J)	E	Bitwise greater than or equal to.
BGT	(I, J)	E	Bitwise greater than.
BIT_SIZE	(I)	I	Number of bits in integer model 13.3.
BLE	(I, J)	E	Bitwise less than or equal to.
BLT	(I, J)	E	Bitwise less than.

Table 13.1: Standard generic intrinsic procedure summary (cont.)

Procedure	Arguments	Class	Description
BTEST	(I, POS)	E	Test single bit in an integer.
CEILING	(A [, KIND])	E	Least integer greater than or equal to A.
CHAR	(I [, KIND])	E	Character from code value.
CMPLX	(X [, KIND]) or (X [, Y, KIND])	E	Conversion to complex type.
COMMAND_ARGUMENT_COUNT	( )	T	Number of command arguments.
CONJG	(Z)	E	Conjugate of a complex number.
COS	(X)	E	Cosine function.
COSH	(X)	E	Hyperbolic cosine function.
COSHAPE	(COARRAY [, KIND])	I	Sizes of codimensions of a coarray.
COUNT	(MASK [, DIM, KIND])	T	Logical array reduced by counting true values.
CPU_TIME	(TIME)	S	Processor time used.
CSHIFT	(ARRAY, SHIFT [, DIM])	T	Circular shift of an array.
DATE_AND_TIME	([DATE, TIME, ZONE, VALUES])	S	Date and time.
DBLE	(A)	E	Conversion to double precision real.
DIGITS	(X)	I	Significant digits in numeric model.
DIM	(X, Y)	E	Maximum of X − Y and zero.
DOT_PRODUCT	(VECTOR_A, VECTOR_B)	T	Dot product of two vectors.
DPROD	(X, Y)	E	Double precision real product.
DSHIFTL	(I, J, SHIFT)	E	Combined left shift.
DSHIFTR	(I, J, SHIFT)	E	Combined right shift.
EOSHIFT	(ARRAY, SHIFT [, BOUNDARY, DIM])	T	End-off shift of the elements of an array.
EPSILON	(X)	I	Model number that is small compared to 1.
ERF	(X)	E	Error function.
ERFC	(X)	E	Complementary error function.
ERFC_SCALED	(X)	E	Scaled complementary error function.
EXECUTE_COMMAND_LINE	(COMMAND [, WAIT, EXITSTAT, CMDSTAT, CMDMSG])	S	Execute a command line.
EXP	(X)	E	Exponential function.
EXPONENT	(X)	E	Exponent of floating-point number.
EXTENDS_TYPE_OF	(A, MOLD)	I	Dynamic type extension inquiry.
FINDLOC	(ARRAY, VALUE, DIM [, MASK, KIND, BACK]) or (ARRAY, VALUE [, MASK, KIND, BACK])	T	Location(s) of a specified value.
FLOOR	(A [, KIND])	E	Greatest integer less than or equal to A.
FRACTION	(X)	E	Fractional part of number.
GAMMA	(X)	E	Gamma function.
GET_COMMAND	([COMMAND, LENGTH, STATUS])	S	Get program invocation command.
GET_COMMAND_ARGUMENT	(NUMBER [, VALUE, LENGTH, STATUS])	S	Get program invocation argument.
GET_ENVIRONMENT_VARIABLE	(NAME [, VALUE, LENGTH, STATUS, TRIM_NAME])	S	Get environment variable.
HUGE	(X)	I	Largest model number.
HYPOT	(X, Y)	E	Euclidean distance function.

Table 13.1: Standard generic intrinsic procedure summary (cont.)

Procedure	Arguments	Class	Description
IACHAR	(C [, KIND])	E	ASCII code value for character.
IALL	(ARRAY, DIM [, MASK]) or (ARRAY [, MASK])	T	Array reduced by IAND function.
IAND	(I, J)	E	Bitwise AND.
IANY	(ARRAY, DIM [, MASK]) or (ARRAY [, MASK])	T	Array reduced by IOR function.
IBCLR	(I, POS)	E	I with bit POS replaced by zero.
IBITS	(I, POS, LEN)	E	Specified sequence of bits.
IBSET	(I, POS)	E	I with bit POS replaced by one.
ICHAR	(C [, KIND])	E	Code value for character.
IEOR	(I, J)	E	Bitwise exclusive OR.
IMAGE_INDEX	(COARRAY, SUB)	I	Image index from cosubscripts.
INDEX	(STRING, SUBSTRING [, BACK, KIND])	E	Character string search.
INT	(A [, KIND])	E	Conversion to integer type.
IOR	(I, J)	E	Bitwise inclusive OR.
IPARITY	(ARRAY, DIM [, MASK]) or (ARRAY [, MASK])	T	Array reduced by IEOR function.
ISHFT	(I, SHIFT)	E	Logical shift.
ISHFTC	(I, SHIFT [, SIZE])	E	Circular shift of the rightmost bits.
IS_CONTIGUOUS	(ARRAY)	I	Array contiguity test (5.5.7).
IS_IOSTAT_END	(I)	E	IOSTAT value test for end of file.
IS_IOSTAT_EOR	(I)	E	IOSTAT value test for end of record.
KIND	(X)	I	Value of the kind type parameter of X.
LBOUND	(ARRAY [, DIM, KIND])	I	Lower bound(s).
LCOBOUND	(COARRAY [, DIM, KIND])	I	Lower cobound(s) of a coarray.
LEADZ	(I)	E	Number of leading zero bits.
LEN	(STRING [, KIND])	I	Length of a character entity.
LEN_TRIM	(STRING [, KIND])	E	Length without trailing blanks.
LGE	(STRING_A, STRING_B)	E	ASCII greater than or equal.
LGT	(STRING_A, STRING_B)	E	ASCII greater than.
LLE	(STRING_A, STRING_B)	E	ASCII less than or equal.
LLT	(STRING_A, STRING_B)	E	ASCII less than.
LOG	(X)	E	Natural logarithm.
LOG_GAMMA	(X)	E	Logarithm of the absolute value of the gamma function.
LOG10	(X)	E	Common logarithm.
LOGICAL	(L [, KIND])	E	Conversion between kinds of logical.
MASKL	(I [, KIND])	E	Left justified mask.
MASKR	(I [, KIND])	E	Right justified mask.
MATMUL	(MATRIX_A, MATRIX_B)	T	Matrix multiplication.
MAX	(A1, A2 [, A3, ...])	E	Maximum value.
MAXEXPONENT	(X)	I	Maximum exponent of a real model.
MAXLOC	(ARRAY, DIM [, MASK, KIND, BACK]) or (ARRAY [, MASK, KIND, BACK])	T	Location(s) of maximum value.
MAXVAL	(ARRAY, DIM [, MASK]) or (ARRAY [, MASK])	T	Maximum value(s) of array.
MERGE	(TSOURCE, FSOURCE, MASK)	E	Expression value selection.
MERGE_BITS	(I, J, MASK)	E	Merge of bits under mask.
MIN	(A1, A2 [, A3, ...])	E	Minimum value.
MINEXPONENT	(X)	I	Minimum exponent of a real model.

Table 13.1: Standard generic intrinsic procedure summary (cont.)

Procedure	Arguments	Class	Description
MINLOC	(ARRAY, DIM [, MASK, KIND, BACK]) or (ARRAY [, MASK, KIND, BACK])	T	Location(s) of minimum value.
MINVAL	(ARRAY, DIM [, MASK]) or (ARRAY [, MASK])	T	Minimum value(s) of array.
MOD	(A, P)	E	Remainder function.
MODULO	(A, P)	E	Modulo function.
MOVE_ALLOC	(FROM, TO)	PS	Move an allocation.
MVBITS	(FROM, FROMPOS, LEN, TO, TOPOS)	ES	Copy a sequence of bits.
NEAREST	(X, S)	E	Adjacent machine number.
NEW_LINE	(A)	I	Newline character.
NINT	(A [, KIND])	E	Nearest integer.
NORM2	(X) or (X, DIM)	T	L_2 norm of an array.
NOT	(I)	E	Bitwise complement.
NULL	([MOLD])	T	Disassociated pointer or unallocated allocatable entity.
NUM_IMAGES	( )	T	Number of images.
OUT_OF_RANGE	(X, MOLD [, ROUND])	E	Whether a value cannot be converted safely.
PACK	(ARRAY, MASK [, VECTOR])	T	Array packed into a vector.
PARITY	(MASK) or (MASK, DIM)	T	Array reduced by .NEQV. operator.
POPCNT	(I)	E	Number of one bits.
POPPAR	(I)	E	Parity expressed as 0 or 1.
PRECISION	(X)	I	Decimal precision of a real model.
PRESENT	(A)	I	Presence of optional argument.
PRODUCT	(ARRAY, DIM [, MASK]) or (ARRAY [, MASK])	T	Array reduced by multiplication.
RADIX	(X)	I	Base of a numeric model.
RANDOM_INIT	(REPEATABLE, IMAGE_-DISTINCT)	S	Initialise the pseudorandom number generator.
RANDOM_NUMBER	(HARVEST)	S	Generate pseudorandom number(s).
RANDOM_SEED	([SIZE, PUT, GET])	S	Restart or query the pseudorandom number generator.
RANGE	(X)	I	Decimal exponent range of a numeric model (13.4).
RANK	(A)	I	Rank of a data object.
REAL	(A [, KIND])	E	Conversion to real type.
REDUCE	(ARRAY, OPERATION, DIM [, MASK, IDENTITY, ORDERED]) or (ARRAY, OPERATION [, MASK, IDENTITY, ORDERED])	T	General reduction of array
REPEAT	(STRING, NCOPIES)	T	Repetitive string concatenation.
RESHAPE	(SOURCE, SHAPE [, PAD, ORDER])	T	Arbitrary shape array construction.
RRSPACING	(X)	E	Reciprocal of relative spacing of model numbers.
SAME_TYPE_AS	(A, B)	I	Dynamic type equality test.
SCALE	(X, I)	E	Real number scaled by radix power.
SCAN	(STRING, SET [, BACK, KIND])	E	Character set membership search.

Table 13.1: Standard generic intrinsic procedure summary (cont.)

Procedure	Arguments	Class	Description
SELECTED_CHAR_KIND	(NAME)	T	Character kind selection.
SELECTED_INT_KIND	(R)	T	Integer kind selection.
SELECTED_REAL_KIND	([P, R, RADIX])	T	Real kind selection.
SET_EXPONENT	(X, I)	E	Real value with specified exponent.
SHAPE	(SOURCE [, KIND])	I	Shape of an array or a scalar.
SHIFTA	(I, SHIFT)	E	Right shift with fill.
SHIFTL	(I, SHIFT)	E	Left shift.
SHIFTR	(I, SHIFT)	E	Right shift.
SIGN	(A, B)	E	Magnitude of A with the sign of B.
SIN	(X)	E	Sine function.
SINH	(X)	E	Hyperbolic sine function.
SIZE	(ARRAY [, DIM, KIND])	I	Size of an array or one extent.
SPACING	(X)	E	Spacing of model numbers (13.4).
SPREAD	(SOURCE, DIM, NCOPIES)	T	Value replicated in a new dimension.
SQRT	(X)	E	Square root.
STORAGE_SIZE	(A [, KIND])	I	Storage size in bits.
SUM	(ARRAY, DIM [, MASK]) or (ARRAY [, MASK])	T	Array reduced by addition.
SYSTEM_CLOCK	([COUNT, COUNT_RATE, COUNT_MAX])	S	Query system clock.
TAN	(X)	E	Tangent function.
TANH	(X)	E	Hyperbolic tangent function.
THIS_IMAGE	( )	T	Index of the invoking image.
THIS_IMAGE	(COARRAY) or (COARRAY, DIM)	T	Cosubscript(s) for this image.
TINY	(X)	I	Smallest positive model number.
TRAILZ	(I)	E	Number of trailing zero bits.
TRANSFER	(SOURCE, MOLD [, SIZE])	T	Transfer physical representation.
TRANSPOSE	(MATRIX)	T	Transpose of an array of rank two.
TRIM	(STRING)	T	String without trailing blanks.
UBOUND	(ARRAY [, DIM, KIND])	I	Upper bound(s).
UCOBOUND	(COARRAY [, DIM, KIND])	I	Upper cobound(s) of a coarray.
UNPACK	(VECTOR, MASK, FIELD)	T	Vector unpacked into an array.
VERIFY	(STRING, SET [, BACK, KIND])	E	Character set non-membership search.

3 The effects of calling COMMAND_ARGUMENT_COUNT, EXECUTE_COMMAND_LINE , GET_COMMAND, and GET_COMMAND_ARGUMENT on any image other than image 1 are processor dependent.

4 If RANDOM_INIT or RANDOM_SEED is called in a segment A, and 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.

5 The use of all other standard intrinsic procedures in unordered segments is subject only to their argument use following the rules in 8.5.2.

13.6 Specific names for standard intrinsic functions

1 Except for 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.

2 A function listed in Table 13.3 is not permitted to be used as an actual argument (12.5.1, C1240), as a target in a procedure pointer assignment statement (7.2.2.2, C730), as an initial target in a procedure declaration statement (12.4.3.7, C1225), or to specify an interface (12.4.3.7, C1221).

Table 13.2: **Unrestricted specific intrinsic functions**

Specific name	Generic name	Argument type and kind
ABS	ABS	default real
ACOS	ACOS	default real
AIMAG	AIMAG	default complex
AINT	AINT	default real
ALOG	LOG	default real
ALOG10	LOG10	default real
AMOD	MOD	default real
ANINT	ANINT	default real
ASIN	ASIN	default real
ATAN	ATAN (X)	default real
ATAN2	ATAN2	default real
CABS	ABS	default complex
CCOS	COS	default complex
CEXP	EXP	default complex
CLOG	LOG	default complex
CONJG	CONJG	default complex
COS	COS	default real
COSH	COSH	default real
CSIN	SIN	default complex
CSQRT	SQRT	default complex
DABS	ABS	double precision real
DACOS	ACOS	double precision real
DASIN	ASIN	double precision real
DATAN	ATAN	double precision real
DATAN2	ATAN2	double precision real
DCOS	COS	double precision real
DCOSH	COSH	double precision real
DDIM	DIM	double precision real
DEXP	EXP	double precision real
DIM	DIM	default real
DINT	AINT	double precision real
DLOG	LOG	double precision real
DLOG10	LOG10	double precision real
DMOD	MOD	double precision real
DNINT	ANINT	double precision real
DPROD	DPROD	default real
DSIGN	SIGN	double precision real
DSIN	SIN	double precision real
DSINH	SINH	double precision real
DSQRT	SQRT	double precision real
DTAN	TAN	double precision real
DTANH	TANH	double precision real
EXP	EXP	default real
IABS	ABS	default integer
IDIM	DIM	default integer
IDNINT	NINT	double precision real
INDEX	INDEX	default character
ISIGN	SIGN	default integer
LEN	LEN	default character
MOD	MOD	default integer
NINT	NINT	default real
SIGN	SIGN	default real
SIN	SIN	default real
SINH	SINH	default real
SQRT	SQRT	default real
TAN	TAN	default real
TANH	TANH	default real

Table 13.3: **Restricted specific intrinsic functions**

Specific name	Generic name	Argument type and kind
AMAX0 (. . .)	REAL (MAX (. . .))	default integer
AMAX1	MAX	default real
AMIN0 (. . .)	REAL (MIN (. . .))	default integer
AMIN1	MIN	default real
CHAR	CHAR	default integer
DMAX1	MAX	double precision real
DMIN1	MIN	double precision real
FLOAT	REAL	default integer
ICHAR	ICHAR	default character
IDINT	INT	double precision real
IFIX	INT	default real
INT	INT	default real
LGE	LGE	default character
LGT	LGT	default character
LLE	LLE	default character
LLT	LLT	default character
MAX0	MAX	default integer
MAX1 (. . .)	INT (MAX (. . .))	default real
MIN0	MIN	default integer
MIN1 (. . .)	INT (MIN (. . .))	default real
REAL	REAL	default integer
SNGL	REAL	double precision real

13.7 Specifications of the standard intrinsic procedures

13.7.1 General

1 Detailed specifications of the standard generic intrinsic procedures are provided in 13.7 in alphabetical order.

2 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.

3 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.

13.7.2 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).

13.7.3 ACHAR (I [, KIND])

1 **Description.** Character from ASCII code value.

2 **Class.** Elemental function.

3 **Arguments.**

I	shall be of type integer.
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.** 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.

6 **Example.** ACHAR (88) has the value 'X'.

13.7.4 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 $0 \leq$ ACOS (X) $\leq \pi$. If it is complex the real part is expressed in radians and lies in the range $0 \leq$ REAL (ACOS (X)) $\leq \pi$.

6 **Example.** ACOS (0.54030231) has the value 1.0 (approximately).

13.7.5 ACOSH (X)

1 **Description.** Inverse hyperbolic cosine 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 cosine function of X. If the result is complex the imaginary part is expressed in radians and lies in the range $0 \leq$ AIMAG (ACOSH (X)) $\leq \pi$

6 **Example.** ACOSH (1.5430806) has the value 1.0 (approximately).

13.7.6 ADJUSTL (STRING)

1 **Description.** Left-justified string value.

2 **Class.** Elemental function.

3 **Argument.** STRING shall be of type character.

4 **Result Characteristics.** Character of the same length and kind type parameter as STRING.

5 **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.

6 **Example.** ADJUSTL (' WORD') has the value 'WORD '.

13.7.7 ADJUSTR (STRING)

1 **Description.** Right-justified string value.

2 **Class.** Elemental function.

3 **Argument.** STRING shall be of type character.

4 **Result Characteristics.** Character of the same length and kind type parameter as STRING.

5 **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.

6 **Example.** ADJUSTR ('WORD ') has the value ' WORD'.

13.7.8 AIMAG (Z)

1 **Description.** Imaginary part of a complex number.

2 **Class.** Elemental function.

3 **Argument.** Z shall be of type complex.

4 **Result Characteristics.** Real with the same kind type parameter as Z.

5 **Result Value.** If Z has the value (x, y), the result has the value y.

6 **Example.** AIMAG ((2.0, 3.0)) has the value 3.0.

13.7.9 AINT (A [, KIND])

1 **Description.** Truncation toward 0 to a whole number.

2 **Class.** Elemental function.

3 **Arguments.**

A	shall be of type real.
KIND (optional)	shall be a scalar integer constant expression.

4 **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.

5 **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.

6 **Examples.** AINT (2.783) has the value 2.0. AINT (-2.783) has the value -2.0.

13.7.10 ALL (MASK) or ALL (MASK, DIM)

1 **Description.** Array reduced by .AND. operator.

2 **Class.** Transformational function.

3 **Arguments.**

MASK shall be a logical array.

DIM shall be an integer scalar with value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of MASK.

4 **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 $[d_1, d_2, \ldots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \ldots, d_n]$ where $[d_1, d_2, \ldots, d_n]$ is the shape of MASK.

5 **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 $(s_1, s_2, \ldots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \ldots, s_n)$ of ALL (MASK, DIM) is equal to ALL (MASK $(s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n)$).

6 **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].

13.7.11 ALLOCATED (ARRAY) or ALLOCATED (SCALAR)

1 **Description.** Allocation status of allocatable variable.

2 **Class.** Inquiry function.

3 **Arguments.**

ARRAY shall be an allocatable array.

SCALAR shall be an allocatable scalar.

4 **Result Characteristics.** Default logical scalar.

5 **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.

13.7.12 ANINT (A [, KIND])

1 **Description.** Nearest whole number.

2 **Class.** Elemental function.

3 **Arguments.**

A shall be of type real.

KIND (optional) shall be a scalar integer constant expression.

4 **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.

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 **Examples.** ANINT (2.783) has the value 3.0. ANINT (−2.783) has the value −3.0.

13.7.13 ANY (MASK) or ANY (MASK, DIM)

1 **Description.** Array reduced by .OR. operator.

2 **Class.** Transformational function.

3 **Arguments.**

MASK shall 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.

4 **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 $[d_1, d_2, \ldots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \ldots, d_n]$ where $[d_1, d_2, \ldots, d_n]$ is the shape of MASK.

5 **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 $(s_1, s_2, \ldots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \ldots, s_n)$ of ANY (MASK, DIM) is equal to ANY (MASK $(s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n)$).

6 **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].

13.7.14 ASIN (X)

1 **Description.** Arcsine (inverse sine) function.

2 **Class.** Elemental function.

3 **Argument.** X shall be of type real with a value that satisfies the inequality $|\text{X}| \leq 1$, 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 arcsin(X). If it is real it is expressed in radians and and lies in the range $-\pi/2 \leq \text{ASIN (X)} \leq \pi/2$. If it is complex the real part is expressed in radians and lies in the range $-\pi/2 \leq \text{REAL (ASIN (X))} \leq \pi/2$.

6 **Example.** ASIN (0.84147098) has the value 1.0 (approximately).

13.7.15 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 $-\pi/2 \leq \text{AIMAG (ASINH (X))} \leq \pi/2$.

6 **Example.** ASINH (1.1752012) has the value 1.0 (approximately).

13.7.16 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 (7.2.2) 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.

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.

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 associated, the targets are not zero-sized storage sequences, and they occupy the same storage units.

Case (vii): If TARGET is present and is an array 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 13.8

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

```
A_PART => A (:N)
```

ASSOCIATED (A_PART, A) is true if N is equal to UBOUND (A, DIM = 1). After the execution of

```
NULLIFY (CUR); NULLIFY (TOP)
```

ASSOCIATED (CUR, TOP) is false.

13.7.17 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).

13.7.18 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 π if Y is positive real zero or the processor cannot 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 $\begin{bmatrix} 3\pi/4 & \pi/4 \\ -3\pi/4 & -\pi/4 \end{bmatrix}$.

13.7.19 ATANH (X)

1 **Description.** Inverse hyperbolic tangent 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 tangent function of X. If the result is complex the imaginary part is expressed in radians and lies in the range $-\pi/2 \leq$ AIMAG (ATANH (X)) $\leq \pi/2$.

6 **Example.** ATANH (0.76159416) has the value 1.0 (approximately).

13.7.20 ATOMIC_DEFINE (ATOM, VALUE)

1 **Description.** Define a variable atomically.

2 **Class.** Atomic subroutine.

3 **Arguments.**

ATOM shall be a scalar coarray or coindexed object and of type integer with kind ATOMIC_INT_KIND or of type logical with kind ATOMIC_LOGICAL_KIND, where ATOMIC_INT_KIND and ATOMIC_LOGICAL_KIND are the named constants in the intrinsic module ISO_FORTRAN_ENV. It is an INTENT (OUT) argument. If its kind is the same as that of VALUE or its type is logical, it becomes defined with the value of VALUE. Otherwise, it becomes defined with the value of INT (VALUE, ATOMIC_INT_KIND).

VALUE shall be scalar and of the same type as ATOM. It is an INTENT (IN) argument.

4 **Example.** CALL ATOMIC_DEFINE (I [3], 4) causes I on image 3 to become defined with the value 4.

13.7.21 ATOMIC_REF (VALUE, ATOM)

1 **Description.** Reference a variable atomically.

2 **Class.** Atomic subroutine.

3 **Arguments.**

VALUE shall be scalar and of the same type as ATOM. It is an INTENT (OUT) argument. If its kind is the same as that of ATOM or its type is logical, it becomes defined with the value of ATOM. Otherwise, it is defined with the value of INT (ATOM, KIND (VALUE)).

ATOM shall be a scalar coarray or coindexed object and of type integer with kind ATOMIC_INT_KIND or of type logical with kind ATOMIC_LOGICAL_KIND, where ATOMIC_INT_KIND and ATOMIC_LOGICAL_KIND are the named constants in the intrinsic module ISO_FORTRAN_ENV. It is an INTENT (IN) argument.

4 **Example.** CALL ATOMIC_REF (VAL, I [3]) causes VAL to become defined with the value of I on image 3.

13.7.22 BESSEL_J0 (X)

1 **Description.** Bessel function of the 1^{st} kind, order 0.

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 Bessel function of the first kind and order zero of X.

6 **Example.** BESSEL_J0 (1.0) has the value 0.765 (approximately).

13.7.23 BESSEL_J1 (X)

1 **Description.** Bessel function of the 1^{st} kind, order 1.

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 Bessel function of the first kind and order one of X.

6 **Example.** BESSEL_J1 (1.0) has the value 0.440 (approximately).

13.7.24 BESSEL_JN (N, X) or BESSEL_JN (N1, N2, X)

1 **Description.** Bessel functions of the 1^{st} kind.

2 **Class.**

Case (i): BESSEL_JN (N,X) is an elemental function.

Case (ii): BESSEL_JN (N1,N2,X) is a transformational function.

3 **Arguments.**

N	shall be of type integer and nonnegative.
N1	an integer scalar with a nonnegative value.
N2	an integer scalar with a nonnegative value.
X	shall be of type real; if the function is transformational, X shall be scalar.

4 **Result Characteristics.** Same type and kind as X.

Case (i): The result of BESSEL_JN (N, X) has the same shape as X.

Case (ii): The result of BESSEL_JN (N1, N2, X) is a rank-one array with extent MAX (N2−N1+1, 0).

5 **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.

6 **Example.** BESSEL_JN (2, 1.0) has the value 0.115 (approximately).

13.7.25 BESSEL_Y0 (X)

1 **Description.** Bessel function of the 2^{nd} kind, order 0.

2 **Class.** Elemental function.

3 **Argument.** X shall be of type real. Its value shall be greater than zero.

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 second kind and order zero of X.

6 **Example.** BESSEL_Y0 (1.0) has the value 0.088 (approximately).

13.7.26 BESSEL_Y1 (X)

1 **Description.** Bessel function of the 2^{nd} kind, order 1.

2 **Class.** Elemental function.

3 **Argument.** X shall be of type real. Its value shall be greater than zero.

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 second kind and order one of X.

6 **Example.** BESSEL_Y1 (1.0) has the value −0.781 (approximately).

13.7.27 BESSEL_YN (N, X) or BESSEL_YN (N1, N2, X)

1 **Description.** Bessel functions of the 2^{nd} kind.

2 **Class.**

Case (i): BESSEL_YN (N, X) is an elemental function.

Case (ii): BESSEL_YN (N1, N2, X) is a transformational function.

3 **Arguments.**

N shall be of type integer and nonnegative.

N1 an integer scalar with a nonnegative value.

N2 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.

4 **Result Characteristics.** Same type and kind as X.

Case (i): The result of BESSEL_YN (N, X) has the same shape as X.

Case (ii): The result of BESSEL_YN (N1, N2, X) is a rank-one array with extent MAX (N2−N1+1, 0).

5 **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.

6 **Example.** BESSEL_YN (2, 1.0) has the value −1.651 (approximately).

13.7.28 BGE (I, J)

1 **Description.** Bitwise greater than or equal to.

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*.

4 **Result Characteristics.** Default logical.

5 **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 13.3.2; otherwise the result is false.

6 The interpretation of a *boz-literal-constant* as a sequence of bits is described in 4.7. The interpretation of an integer value as a sequence of bits is described in 13.3.

7 **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.

13.7.29 BGT (I, J)

1 **Description.** Bitwise greater than.

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*.

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 13.3.2; otherwise the result is false.

6 The interpretation of a *boz-literal-constant* as a sequence of bits is described in 4.7. The interpretation of an integer value as a sequence of bits is described in 13.3.

7 **Example.** BGT (Z'FF', Z'FC') has the value true. BGT (0, −1) has the value false.

13.7.30 BIT_SIZE (I)

1 **Description.** Number of bits in integer model 13.3.

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 13.3.

6 **Example.** BIT_SIZE (1) has the value 32 if z of the model is 32.

13.7.31 BLE (I, J)

1 **Description.** Bitwise less than or equal to.

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*.

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 13.3.2; otherwise the result is false.

6 The interpretation of a *boz-literal-constant* as a sequence of bits is described in 4.7. The interpretation of an integer value as a sequence of bits is described in 13.3.

7 **Example.** BLE (0, J) has the value true for any value of J. BLE (−1, 0) has the value false.

13.7.32 BLT (I, J)

1 **Description.** Bitwise less than.

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*.

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 13.3.2; otherwise the result is false.

6 The interpretation of a *boz-literal-constant* as a sequence of bits is described in 4.7. The interpretation of an integer value as a sequence of bits is described in 13.3.

7 **Example.** BLT (0, −1) has the value true. BLT (Z'FF', Z'FC') has the value false.

13.7.33 BTEST (I, POS)

1 **Description.** Test single bit in an integer.

2 **Class.** Elemental function.

3 **Arguments.**

I	shall be of type integer.
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 13.3.

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 $\begin{bmatrix} \text{true} & \text{false} \\ \text{false} & \text{false} \end{bmatrix}$.

13.7.34 CEILING (A [, KIND])

1 **Description.** Least integer greater than or equal to A.

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 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.

13.7.35 CHAR (I [, KIND])

1 **Description.** Character from code value.

2 **Class.** Elemental function.

3 **Arguments.**

I	shall be of type integer with a value in the range $0 \leq \text{I} \leq n-1$, where n is the number of characters in the collating sequence associated with the specified kind type parameter.
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. ICHAR (CHAR (I, KIND (C))) shall have the value I for $0 \leq \text{I} \leq n - 1$ and CHAR (ICHAR (C), KIND (C)) shall have the value C for any character C capable of representation in the processor.

6 **Example.** CHAR (88) has the value 'X' on a processor using the ASCII collating sequence for default characters.

13.7.36 CMPLX (X [, KIND]) or (X [, Y, KIND])

1 **Description.** Conversion to complex type.

2 **Class.** Elemental function.

3 **Arguments for CMPLX(X [, KIND]).**

X shall be of type complex.

KIND (optional) shall be a scalar integer constant expression.

4 **Arguments for CMPLX(X [, Y, KIND]).**

X shall be of type integer or real, or a *boz-literal-constant*.

Y (optional) shall be of type integer or real, or a *boz-literal-constant*.

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 KIND (0.0). If X is complex, the result is the same as that of CMPLX (REAL (X), AIMAG (X), KIND). The result of CMPLX (X, Y, KIND) has the complex value whose real part is REAL (X, KIND) and whose imaginary part is REAL (Y, KIND).

7 **Example.** CMPLX (−3) has the value (−3.0, 0.0).

13.7.37 COMMAND_ARGUMENT_COUNT ( )

1 **Description.** Number of command arguments.

2 **Class.** Transformational function.

3 **Argument.** None.

4 **Result Characteristics.** Default integer scalar.

5 **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.

6 **Example.** See 13.7.67.

13.7.38 CONJG (Z)

1 **Description.** Conjugate of a complex number.

2 **Class.** Elemental function.

3 **Argument.** Z shall be of type complex.

4 **Result Characteristics.** Same as Z.

5 **Result Value.** If Z has the value (x, y), the result has the value $(x, -y)$.

6 **Example.** CONJG ((2.0, 3.0)) has the value (2.0, −3.0).

13.7.39 COS (X)

1 **Description.** Cosine 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 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.

6 **Example.** COS (1.0) has the value 0.54030231 (approximately).

13.7.40 COSH (X)

1 **Description.** Hyperbolic cosine 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 cosh(X). If X is of type complex its imaginary part is regarded as a value in radians.

6 **Example.** COSH (1.0) has the value 1.5430806 (approximately).

13.7.41 COSHAPE (COARRAY [, KIND])

1 **Description.** Sizes of codimensions of a coarray.

2 **Class.** Inquiry function.

3 **Arguments.**

COARRAY shall be a coarray of any type. It shall not be an unallocated allocatable coarray.

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 corank of COARRAY.

5 **Result Value.** The result has a value whose i^{th} element is equal to the size of the i^{th} codimension of COARRAY, as given by UCOBOUND(COARRAY,i)−LCOBOUND(COARRAY,i)+1.

6 **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.

```
REAL, ALLOCATABLE :: C[:,:], D[:,:]
INTEGER, ALLOCATABLE :: COSHAPE_C(:)
```

```
...
COSHAPE_C = COSHAPE(C)
ALLOCATE ( D[COSHAPE_C(1),*] )
```

13.7.42 COUNT (MASK [, DIM, KIND])

1 **Description.** Logical array reduced by counting true values.

2 **Class.** Transformational function.

3 **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.

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 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.

5 **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 $(s_1, s_2, \ldots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \ldots, s_n)$ of the result is equal to the number of true elements of MASK $(s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n)$.

6 **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 /= C, DIM = 1) is [2, 0, 1] and COUNT (B /= C, DIM = 2) is [1, 2].

13.7.43 CPU_TIME (TIME)

1 **Description.** Processor time used.

2 **Class.** Subroutine.

3 **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.

4 **Example.**

```
REAL T1, T2
...
CALL CPU_TIME(T1)
...   ! Code to be timed.
CALL CPU_TIME(T2)
WRITE (*,*) 'Time taken by code was ', T2-T1, ' seconds'
```

writes the processor time taken by a piece of code.

NOTE 13.9

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.

13.7.44 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 $[d_1, d_2, \ldots, d_{\mathrm{DIM}-1}, d_{\mathrm{DIM}+1}, \ldots, d_n]$ where $[d_1, d_2, \ldots, d_n]$ is the shape of ARRAY.

DIM (optional) shall be an integer scalar with a value in the range $1 \leq \mathrm{DIM} \leq n$, 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 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 $(s_1, s_2, \ldots, s_{\mathrm{DIM}-1}, :, s_{\mathrm{DIM}+1}, \ldots, s_n)$ of the result has a value equal to CSHIFT (ARRAY $(s_1, s_2, \ldots, s_{\mathrm{DIM}-1}, :, s_{\mathrm{DIM}+1}, \ldots, s_n)$, sh, 1), where sh is SHIFT or SHIFT $(s_1, s_2, \ldots, s_{\mathrm{DIM}-1}, s_{\mathrm{DIM}+1}, \ldots, s_n)$.

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 $\begin{bmatrix} 3 & 1 & 2 \\ 5 & 6 & 4 \\ 7 & 8 & 9 \end{bmatrix}$.

13.7.45 DATE_AND_TIME ([DATE, TIME, ZONE, VALUES])

1 **Description.** Date and time.

2 **Class.** Subroutine.

3 **Arguments.**

DATE (optional) shall be a default character scalar. It is an INTENT (OUT) argument. It is assigned a value of the form *CCYYMMDD*, where *CC* is the century, *YY* is the year within the century, *MM* is the month within the year, and *DD* is the day within the month. If there is no date available, DATE is assigned all blanks.

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. If there is no clock available, TIME is assigned all blanks.

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. If this information is not available, ZONE is assigned all blanks.

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.**

6
```
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)
```

7 If run in Geneva, Switzerland on April 12, 2008 at 15:27:35.5 with a system configured for the local time zone, this sample 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.

NOTE 13.10

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).

13.7.46 DBLE (A)

1 **Description.** Conversion to double precision real.

2 **Class.** Elemental function.

3 **Argument.** A shall be of type integer, real, complex, or a *boz-literal-constant*.

4 **Result Characteristics.** Double precision real.

5 **Result Value.** The result has the value REAL (A, KIND (0.0D0)).

6 **Example.** DBLE (−3) has the value −3.0D0.

13.7.47 DIGITS (X)

1 **Description.** Significant digits in 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 q if X is of type integer and p if X is of type real, where q and p are as defined in 13.4 for the model representing numbers of the same type and kind type parameter as X.

6 **Example.** DIGITS (X) has the value 24 for real X whose model is as in Note 13.5.

13.7.48 DIM (X, Y)

1 **Description.** Maximum of X − Y and zero.

2 **Class.** Elemental function.

3 **Arguments.**

X	shall be of type integer or real.
Y	shall be of the same type and kind type parameter as X.

4 **Result Characteristics.** Same as X.

5 **Result Value.** The value of the result is the maximum of X − Y and zero.

6 **Example.** DIM (−3.0, 2.0) has the value 0.0.

13.7.49 DOT_PRODUCT (VECTOR_A, VECTOR_B)

1 **Description.** Dot product of two vectors.

2 **Class.** Transformational function.

3 **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.

4 **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 7.1.9.3. 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 7.1.9.3. The result is scalar.

5 **Result Value.**

Case (i): If VECTOR_A is of type integer or real, the result has the value SUM (VECTOR_A*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 SUM (CONJG (VECTOR_A)*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 ANY (VECTOR_A .AND. VECTOR_B). If the vectors have size zero, the result has the value false.

6 **Example.** DOT_PRODUCT ([1, 2, 3], [2, 3, 4]) has the value 20.

13.7.50 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.

13.7.51 DSHIFTL (I, J, SHIFT)

1 **Description.** Combined left 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-constant*s.

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 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 13.3.

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).

13.7.52 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-constant*s.
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 13.3.

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).

13.7.53 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 $[d_1, d_2, \ldots, d_{\mathrm{DIM}-1}, d_{\mathrm{DIM}+1}, \ldots, d_n]$ 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 $[d_1, d_2, \ldots, d_{\mathrm{DIM}-1}, d_{\mathrm{DIM}+1}, \ldots, d_n]$. BOUNDARY is permitted to be absent only for the types in Table 13.4, 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.

Table 13.4: **Default BOUNDARY values for EOSHIFT**

Type of ARRAY	Value of BOUNDARY
Integer	0
Real	0.0
Complex	(0.0, 0.0)
Logical	.FALSE.
Character (*len*)	*len* blanks

DIM (optional)	shall be an integer scalar with a value in the range $1 \leq \mathrm{DIM} \leq n$, 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 ARRAY $(s_1, s_2, \ldots, s_{\mathrm{DIM}-1}, s_{\mathrm{DIM}} + sh,$

$s_{\mathrm{DIM}+1}, \ldots, s_n$) where sh is SHIFT or SHIFT ($s_1, s_2, \ldots, s_{\mathrm{DIM}-1}, s_{\mathrm{DIM}+1}, \ldots, s_n$) provided the inequality LBOUND (ARRAY, DIM) $\leq s_{\mathrm{DIM}} + sh \leq$ UBOUND (ARRAY, DIM) holds and is otherwise BOUNDARY or BOUNDARY ($s_1, s_2, \ldots, s_{\mathrm{DIM}-1}, s_{\mathrm{DIM}+1}, \ldots, s_n$).

6 **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 $\begin{bmatrix} \mathrm{A} & \mathrm{B} & \mathrm{C} \\ \mathrm{D} & \mathrm{E} & \mathrm{F} \\ \mathrm{G} & \mathrm{H} & \mathrm{I} \end{bmatrix}$, then the value of EOSHIFT (M, SHIFT = −1, BOUNDARY = '*', DIM = 2) is $\begin{bmatrix} * & \mathrm{A} & \mathrm{B} \\ * & \mathrm{D} & \mathrm{E} \\ * & \mathrm{G} & \mathrm{H} \end{bmatrix}$, and the value of EOSHIFT (M, SHIFT = [−1, 1, 0], BOUNDARY = ['*', '/', '?'], DIM = 2) is $\begin{bmatrix} * & \mathrm{A} & \mathrm{B} \\ \mathrm{E} & \mathrm{F} & / \\ \mathrm{G} & \mathrm{H} & \mathrm{I} \end{bmatrix}$.

13.7.54 EPSILON (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 13.4 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 13.5.

13.7.55 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, $\frac{2}{\sqrt{\pi}} \int_0^X \exp(-t^2)\,\mathrm{d}t$.

6 **Example.** ERF (1.0) has the value 0.843 (approximately).

13.7.56 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, $1 - \mathrm{ERF}$ (X); this is equivalent to $\frac{2}{\sqrt{\pi}} \int_X^{\infty} \exp(-t^2) \mathrm{d}t$.

6 **Example.** ERFC (1.0) has the value 0.157 (approximately).

13.7.57 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, $\exp(X^2) \frac{2}{\sqrt{\pi}} \int_X^{\infty} \exp(-t^2) \, \mathrm{d}t$.

6 **Example.** ERFC_SCALED (20.0) has the value 0.02817434874 (approximately).

NOTE 13.11

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 $X >$ HUGE (X)$/\sqrt{\pi}$.

13.7.58 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 condition 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.

13.7.59 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).

13.7.60 EXPONENT (X)

1 **Description.** Exponent of floating-point number.

2 **Class.** Elemental function.

3 **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 (13.4), 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 13.5.

13.7.61 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 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 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 13.12

The dynamic type of a disassociated pointer or unallocated allocatable variable is its declared type.

13.7.62 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 7.2 for the operator == or the operator .EQV..
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): 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 $(s_1, s_2, \ldots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \ldots, s_n)$ of the result is equal to FINDLOC (ARRAY $(s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n)$, VALUE, DIM=1 [, MASK = MASK $(s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n)$]).

6 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.

7 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.

8 **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.

13.7.63 FLOOR (A [, KIND])

1 **Description.** Greatest integer less than or equal to A.

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 has a value equal to the greatest integer less than or equal to A.

6 **Examples.** FLOOR (3.7) has the value 3. FLOOR (−3.7) has the value −4.

13.7.64 FRACTION (X)

1 **Description.** Fractional part of number.

2 **Class.** Elemental function.

3 **Argument.** X shall be of type real.

4 **Result Characteristics.** Same as X.

5 **Result Value.** The result has the value X $\times\, b^{-e}$, where b and e are as defined in 13.4 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.

6 **Example.** FRACTION (3.0) has the value 0.75 for reals whose model is as in Note 13.5.

13.7.65 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,

$$\Gamma(X) = \begin{cases} \int_0^\infty t^{X-1} \exp(-t)\,\mathrm{d}t & X > 0 \\ \int_0^\infty t^{X-1} \left(\exp(-t) - \sum_{k=0}^{n} \frac{(-t)^k}{k!}\right) \mathrm{d}t & -n-1 < X < -n,\ n \text{ an integer } \geq 0 \end{cases}$$

6 **Example.** GAMMA (1.0) has the value 1.000 (approximately).

13.7.66 GET_COMMAND ([COMMAND, LENGTH, STATUS])

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.

13.7.67 GET_COMMAND_ARGUMENT (NUMBER [, VALUE, LENGTH, STATUS])

1 **Description.** Get program invocation argument.

2 **Class.** Subroutine.

3 **Arguments.**

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.

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.

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.

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.

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.

4 **Example.**

```
PROGRAM echo
```

```
  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 echo
```

13.7.68 GET_ENVIRONMENT_VARIABLE (NAME [, VALUE, LENGTH, STATUS, TRIM_NAME])

1 **Description.** Get environment variable.

2 **Class.** Subroutine.

3 **Arguments.**

NAME shall be a default character scalar. It is an INTENT (IN) argument. The interpretation of case is processor dependent

VALUE (optional) shall be a default character scalar. It is an INTENT (OUT) argument. It is assigned the value of the environment variable specified by NAME. 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.

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 specified environment variable exists and has a value, LENGTH is set to the length of that value. Otherwise LENGTH is set to 0.

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 the environment variable exists and either has no value, its value is successfully assigned to VALUE, or the VALUE argument is not present, STATUS is set to zero. STATUS is set to -1 if the 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.

TRIM_NAME (optional) shall be a logical scalar. It is an INTENT (IN) argument. If TRIM_NAME is present with the value false then trailing blanks in NAME are considered significant if the processor supports trailing blanks in environment variable names. Otherwise trailing blanks in NAME are not considered part of the environment variable's name.

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.

Unresolved Technical Issue 011

> **Missing example for GET_ENVIRONMENT_VARIABLE.**
>
> We ought to have at least one example for each intrinsic.

13.7.69 HUGE (X)

1 **Description.** Largest model number.

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.** Scalar of the same type and kind type parameter as X.

5 **Result Value.** The result has the value $r^q - 1$ if X is of type integer and $(1 - b^{-p})b^{e_{\max}}$ if X is of type real, where r, q, b, p, and $e_{\max}$ are as defined in 13.4 for the model representing numbers of the same type and kind type parameter as X.

6 **Example.** HUGE (X) has the value $(1 - 2^{-24}) \times 2^{127}$ for real X whose model is as in Note 13.5.

13.7.70 HYPOT (X, Y)

1 **Description.** Euclidean distance function.

2 **Class.** Elemental function.

3 **Arguments.**

X shall be of type real.

Y shall be of type real with the same kind type parameter as X.

4 **Result Characteristics.** Same as X.

5 **Result Value.** The result has a value equal to a processor-dependent approximation to the Euclidean distance, $\sqrt{\mathrm{X}^2 + \mathrm{Y}^2}$, without undue overflow or underflow.

6 **Example.** HYPOT (3.0, 4.0) has the value 5.0 (approximately).

13.7.71 IACHAR (C [, KIND])

1 **Description.** ASCII code value for character.

2 **Class.** Elemental function.

3 **Arguments.**

C shall be of type character and of length one.

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.** 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 non-negative 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.

6 **Example.** IACHAR ('X') has the value 88.

13.7.72 IALL (ARRAY, DIM [, MASK]) or IALL (ARRAY [, MASK])

1 **Description.** Array reduced by IAND function.

2 **Class.** Transformational function.

3 **Arguments.**

ARRAY shall be an array of type integer.

DIM shall be an integer scalar with a value in the range $1 \leq \mathrm{DIM} \leq n$, 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 or if ARRAY has rank one; otherwise, the result is an array of rank $n-1$ and shape $[d_1, d_2, \ldots, d_{\mathrm{DIM}-1}, d_{\mathrm{DIM}+1}, \ldots, d_n]$ where $[d_1, d_2, \ldots, d_n]$ is the shape of ARRAY.

5 **Result Value.**

Case (i): 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 $(s_1, s_2, \ldots, s_{\mathrm{DIM}-1}, s_{\mathrm{DIM}+1}, \ldots, s_n)$ of the result is equal to IALL (ARRAY $(s_1, s_2, \ldots, s_{\mathrm{DIM}-1}, :, s_{\mathrm{DIM}+1}, \ldots, s_n)$ [, MASK = MASK $(s_1, s_2, \ldots, s_{\mathrm{DIM}-1}, :, s_{\mathrm{DIM}+1}, \ldots, s_n)$]).

6 **Examples.** IALL ([14, 13, 11]) has the value 8. IALL ([14, 13, 11], MASK=[.true., .false., .true]) has the value 10.

13.7.73 IAND (I, J)

1 **Description.** Bitwise AND.

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-constant*s.

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 result has the value obtained by combining I and J bit-by-bit according to the following truth table:

I	J	IAND (I, J)
1	1	1
1	0	0
0	1	0
0	0	0

6 The model for the interpretation of an integer value as a sequence of bits is in 13.3.

7 **Example.** IAND (1, 3) has the value 1.

13.7.74 IANY (ARRAY, DIM [, MASK]) or IANY (ARRAY [, MASK])

1 **Description.** Reduce array with bitwise OR operation.

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 $1 \leq DIM \leq n$, 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 or if ARRAY has rank one; otherwise, the result is an array of rank $n - 1$ and shape $[d_1, d_2, \ldots, d_{\mathrm{DIM}-1}, d_{\mathrm{DIM}+1}, \ldots, d_n]$ where $[d_1, d_2, \ldots, d_n]$ is the shape of ARRAY.

5 **Result Value.**

Case (i): The result of 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 IANY (ARRAY, MASK=MASK) has a value equal to IANY (PACK (ARRAY, MASK)).

Case (iii): The result of IANY (ARRAY, DIM=DIM [, MASK=MASK]) has a value equal to that of IANY (ARRAY [, MASK=MASK]) if ARRAY has rank one. Otherwise, the value of element $(s_1, s_2, \ldots, s_{\mathrm{DIM}-1}, s_{\mathrm{DIM}+1}, \ldots, s_n)$ of the result is equal to IANY (ARRAY $(s_1, s_2, \ldots, s_{\mathrm{DIM}-1}, :, s_{\mathrm{DIM}+1}, \ldots, s_n)$ [, MASK = MASK $(s_1, s_2, \ldots, s_{\mathrm{DIM}-1}, :, s_{\mathrm{DIM}+1}, \ldots, s_n)$]).

6 **Examples.** IANY ([14, 13, 8]) has the value 15. IANY ([14, 13, 8], MASK=[.true., .false., .true]) has the value 14.

13.7.75 IBCLR (I, POS)

1 **Description.** I with bit POS replaced by zero.

2 **Class.** Elemental function.

3 **Arguments.**

I shall be of type integer.

POS shall be of type integer. It shall be nonnegative and less than BIT_SIZE (I).

4 **Result Characteristics.** Same as I.

5 **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 13.3.

6 **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].

13.7.76 IBITS (I, POS, LEN)

1 **Description.** Specified sequence of bits.

2 **Class.** Elemental function.

3 **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.

4 **Result Characteristics.** Same as I.

5 **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 13.3.

6 **Example.** IBITS (14, 1, 3) has the value 7.

13.7.77 IBSET (I, POS)

1 **Description.** I with bit POS replaced by one.

2 **Class.** Elemental function.

3 **Arguments.**

I shall be of type integer.

POS shall be of type integer. It shall be nonnegative and less than BIT_SIZE (I).

4 **Result Characteristics.** Same as I.

5 **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 13.3.

6 **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].

13.7.78 ICHAR (C [, KIND])

1 **Description.** Code value for character.

2 **Class.** Elemental function.

3 **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.

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 position of C in the processor collating sequence associated with the kind type parameter of C; it is non-negative 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 <= D is true if and only if ICHAR (C) <= ICHAR (D) is true and C == D is true if and only if ICHAR (C) == ICHAR (D) is true.

6 **Example.** ICHAR ('X') has the value 88 on a processor using the ASCII collating sequence for default characters.

13.7.79 IEOR (I, J)

1 **Description.** Bitwise exclusive 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-constant*s.

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 result has the value obtained by combining I and J bit-by-bit according to the following truth table:

I	J	IEOR (I, J)
1	1	0
1	0	1
0	1	1
0	0	0

6 The model for the interpretation of an integer value as a sequence of bits is in 13.3.

7 **Example.** IEOR (1, 3) has the value 2.

13.7.80 IMAGE_INDEX (COARRAY, SUB)

1 **Description.** Image index from cosubscripts.

2 **Class.** Inquiry function.

3 **Arguments.**

COARRAY shall be a coarray of any type.

SUB shall be a rank-one integer array of size equal to the corank of COARRAY.

4 **Result Characteristics.** Default integer scalar.

5 **Result Value.** If the value of SUB is a valid sequence of cosubscripts for COARRAY, the result is the index of the corresponding image. 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).

13.7.81 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 $1 \leq$ I $\leq$ 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.

13.7.82 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 $|A| \geq 1$, INT (A) is the integer whose magnitude is the largest integer that does not exceed the magnitude of A and whose sign is the same as the sign of A.

Case (iii): If A is of type complex, INT (A) = 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 13.3 is the same as that of A as modified by padding or truncation according to 13.3.3. The interpretation of a bit sequence whose most significant bit is 1 is processor dependent.

6 **Example.** INT (-3.7) has the value -3.

13.7.83 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-constant*s.

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 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 truth table:

I	J	IOR (I, J)
1	1	1
1	0	1
0	1	1
0	0	0

7 The model for the interpretation of an integer value as a sequence of bits is in 13.3.

8 **Example.** IOR (5, 3) has the value 7.

13.7.84 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=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 $(s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n)$ [, MASK = MASK $(s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n)$]).

6 **Examples.** IPARITY ([14, 13, 8]) has the value 11. IPARITY ([14, 13, 8], MASK=[.true., .false., .true]) has the value 6.

13.7.85 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.

5 **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 13.3.

6 **Example.** ISHFT (3, 1) has the value 6.

13.7.86 ISHFTC (I, SHIFT [, SIZE])

1 **Description.** Circular shift of the rightmost bits.

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 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).

4 **Result Characteristics.** Same as I.

5 **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 13.3.

6 **Example.** ISHFTC (3, 2, 3) has the value 5.

13.7.87 IS_CONTIGUOUS (ARRAY)

1 **Description.** Array contiguity test (5.5.7).

2 **Class.** Inquiry function.

3 **Argument.** ARRAY may be of any type. It shall be assumed-rank or an array. If it is a pointer it shall be associated.

4 **Result Characteristics.** Default logical scalar.

5 **Result Value.** The result has the value true if ARRAY has rank zero or is contiguous, and false otherwise.

6 **Example.** After the pointer assignment AP => TARGET (1:10:2), IS_CONTIGUOUS (AP) has the value false.

13.7.88 IS_IOSTAT_END (I)

1 **Description.** IOSTAT value test for end of file.

2 **Class.** Elemental function.

3 **Argument.** I shall be of type integer.

4 **Result Characteristics.** Default logical.

5 **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 (9.11.5) that would indicate an end-of-file condition.

13.7.89 IS_IOSTAT_EOR (I)

1 **Description.** IOSTAT value test for end of record.

2 **Class.** Elemental function.

3 **Argument.** I shall be of type integer.

4 **Result Characteristics.** Default logical.

5 **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 (9.11.5) that would indicate an end-of-record condition.

13.7.90 KIND (X)

1 **Description.** Value of the kind type parameter of X.

2 **Class.** Inquiry function.

3 **Argument.** X may be of any intrinsic type. It may be a scalar or an array.

4 **Result Characteristics.** Default integer scalar.

5 **Result Value.** The result has a value equal to the kind type parameter value of X.

6 **Example.** KIND (0.0) has the kind type parameter value of default real.

13.7.91 LBOUND (ARRAY [, DIM, KIND])

1 **Description.** Lower bound(s).

2 **Class.** Inquiry function.

3 **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.

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 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 13.13

If ARRAY is assumed-rank and has rank zero, DIM cannot be present since it cannot satisfy the requirement $1 \leq \text{DIM} \leq 0$.

6 **Examples.** If A is declared by the statement

7
```
REAL A (2:3, 7:10)
```

8 then LBOUND (A) is [2, 7] and LBOUND (A, DIM=2) is 7.

13.7.92 LCOBOUND (COARRAY [, DIM, KIND])

1 **Description.** Lower cobound(s) of a coarray.

2 **Class.** Inquiry function.

3 **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 $1 \leq \text{DIM} \leq n$, 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.

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 corank of COARRAY.

5 **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.

6 **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.

13.7.93 LEADZ (I)

1 **Description.** Number of leading zero bits.

2 **Class.** Elemental function.

3 **Argument.** I shall be of type integer.

4 **Result Characteristics.** Default integer.

5 **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 13.3.

6 **Examples.** LEADZ (1) has the value 31 if BIT_SIZE (1) has the value 32.

13.7.94 LEN (STRING [, KIND])

1 **Description.** Length of a character entity.

2 **Class.** Inquiry function.

3 **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.

4 **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.

5 **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.

6 **Example.** If C is declared by the statement

7
```
CHARACTER (11) C (100)
```

8 LEN (C) has the value 11.

13.7.95 LEN_TRIM (STRING [, KIND])

1 **Description.** Length without trailing blanks.

2 **Class.** Elemental function.

3 **Arguments.**

STRING shall be of type character.

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 number of characters remaining after any trailing blanks in STRING are removed. If the argument contains no nonblank characters, the result is zero.

6 **Examples.** LEN_TRIM (' A B ') has the value 4 and LEN_TRIM ('   ') has the value 0.

13.7.96 LGE (STRING_A, STRING_B)

1 **Description.** ASCII greater than or equal.

2 **Class.** Elemental function.

3 **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.

4 **Result Characteristics.** Default logical.

5 **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.

NOTE 13.14

The result is true if both STRING_A and STRING_B are of zero length.

6 **Example.** LGE ('ONE', 'TWO') has the value false.

13.7.97 LGT (STRING_A, STRING_B)

1 **Description.** ASCII greater than.

2 **Class.** Elemental function.

3 **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.

4 **Result Characteristics.** Default logical.

5 **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 13.15

The result is false if both STRING_A and STRING_B are of zero length.

6 **Example.** LGT ('ONE', 'TWO') has the value false.

13.7.98 LLE (STRING_A, STRING_B)

1 **Description.** ASCII less than or equal.

2 **Class.** Elemental function.

3 **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.

4 **Result Characteristics.** Default logical.

5 **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 13.16

The result is true if both STRING_A and STRING_B are of zero length.

6 **Example.** LLE ('ONE', 'TWO') has the value true.

13.7.99 LLT (STRING_A, STRING_B)

1 **Description.** ASCII less than.

2 **Class.** Elemental function.

3 **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.

4 **Result Characteristics.** Default logical.

5 **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 13.17

The result is false if both STRING_A and STRING_B are of zero length.

6 **Example.** LLT ('ONE', 'TWO') has the value true.

13.7.100 LOG (X)

1 **Description.** Natural logarithm.

2 **Class.** Elemental function.

3 **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.

4 **Result Characteristics.** Same as X.

5 **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 $-\pi \le \omega \le \pi$. 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 cannot distinguish between positive and negative real zero, and approximately $-\pi$ if the imaginary part of X is negative real zero.

6 **Example.** LOG (10.0) has the value 2.3025851 (approximately).

13.7.101 LOG_GAMMA (X)

1 **Description.** Logarithm of the absolute value of the 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 natural logarithm of the absolute value of the gamma function of X.

6 **Example.** LOG_GAMMA (3.0) has the value 0.693 (approximately).

13.7.102 LOG10 (X)

1 **Description.** Common logarithm.

2 **Class.** Elemental function.

3 **Argument.** X shall be of type real. The value of X shall be greater than zero.

4 **Result Characteristics.** Same as X.

5 **Result Value.** The result has a value equal to a processor-dependent approximation to $\log_{10}$X.

6 **Example.** LOG10 (10.0) has the value 1.0 (approximately).

13.7.103 LOGICAL (L [, KIND])

1 **Description.** Conversion between kinds of logical.

2 **Class.** Elemental function.

3 **Arguments.**

L	shall be of type logical.
KIND (optional)	shall be a scalar integer constant expression.

4 **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.

5 **Result Value.** The value is that of L.

6 **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.

13.7.104 MASKL (I [, KIND])

1 **Description.** Left justified mask.

2 **Class.** Elemental function.

3 **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 13.3 for the kind of the result.

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 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 13.3.

6 **Example.** MASKL (3) has the value SHIFTL (7, BIT_SIZE (0) − 3).

13.7.105 MASKR (I [, KIND])

1 **Description.** Right justified mask.

2 **Class.** Elemental function.

3 **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 13.3 for the kind of the result.

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 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 13.3.

6 **Example.** MASKR (3) has the value 7.

13.7.106 MATMUL (MATRIX_A, MATRIX_B)

1 **Description.** Matrix multiplication.

2 **Class.** Transformational function.

3 **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.

4 **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 7.1.9.3 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 7.1.9.3 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 $[n]$.

5 **Result Value.**

Case (i): Element (i, j) of the result has the value SUM (MATRIX_A (i, :) * MATRIX_B (:, j)) if the arguments are of numeric type and has the value ANY (MATRIX_A (i, :) .AND. MATRIX_B (:, j)) if the arguments are of logical type.

Case (ii): Element (j) of the result has the value SUM (MATRIX_A (:) * MATRIX_B (:, j)) if the arguments are of numeric type and has the value ANY (MATRIX_A (:) .AND. MATRIX_B (:, j)) if the arguments are of logical type.

Case (iii): Element (i) of the result has the value SUM (MATRIX_A (i, :) * MATRIX_B (:)) if the arguments are of numeric type and has the value ANY (MATRIX_A (i, :) .AND. MATRIX_B (:)) if the arguments are of logical type.

6 **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 matrix-vector product AY with the value [14, 20].

13.7.107 MAX (A1, A2 [, A3, ...])

1 **Description.** Maximum value.

2 **Class.** Elemental function.

3 **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.

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 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.

6 **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 '].

13.7.108 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 $e_{\max}$, as defined in 13.4 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 13.5.

13.7.109 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 $(s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n)$,
DIM = 1
[, MASK = MASK $(s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n)$,
KIND = KIND,
BACK = BACK]).

6 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.

7 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.

8 **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.

13.7.110 MAXVAL (ARRAY, DIM [, MASK]) or MAXVAL (ARRAY [, MASK])

1 **Description.** Maximum value(s) of array.

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 $1 \le \text{DIM} \le n$, 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 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 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 $(s_1, s_2, \ldots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \ldots, s_n)$ of the result is equal to

MAXVAL (ARRAY $(s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n)$
[, MASK = MASK $(s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n)$]).

6 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.

7 **Examples.**

Case (i): The value of MAXVAL ([1, 2, 3]) is 3.

Case (ii): MAXVAL (C, MASK = C < 0.0) is the maximum of the negative elements of C.

Case (iii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 7 & 6 \end{bmatrix}$, MAXVAL (B, DIM = 1) is [2, 7, 6] and MAXVAL (B, DIM = 2) is [5, 7].

13.7.111 MERGE (TSOURCE, FSOURCE, MASK)

1 **Description.** Expression value selection.

2 **Class.** Elemental function.

3 **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.

4 **Result Characteristics.** Same as TSOURCE.

5 **Result Value.** The result is TSOURCE if MASK is true and FSOURCE otherwise.

6 **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 $\begin{bmatrix} \text{T} & . & \text{T} \\ . & . & \text{T} \end{bmatrix}$, 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.

13.7.112 MERGE_BITS (I, J, MASK)

1 **Description.** Merge of bits under mask.

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-constant*s.
MASK	shall be of type integer or a *boz-literal-constant*. If MASK is of type integer, it shall have the same kind type parameter as each other argument of type integer.

4 **Result Characteristics.** Same as I if I is of type integer; otherwise, same as J.

5 **Result Value.** If any argument is a *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))).

6 **Example.** MERGE_BITS (13, 18, 22) has the value 4.

13.7.113 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 '].

13.7.114 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 $e_{\min}$, as defined in 13.4 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 13.5.

13.7.115 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.**

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 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 $(s_1, s_2, \ldots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \ldots, s_n)$ of the result is equal to

MINLOC (ARRAY $(s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n)$,
DIM = 1
[, MASK = MASK $(s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n)$,
KIND = KIND,
BACK = BACK]).

6 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.

7 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.

8 **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 $\begin{bmatrix} 0 & -5 & 8 & -3 \\ 3 & 4 & -1 & 2 \\ 1 & 5 & 6 & -4 \end{bmatrix}$, 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 $\begin{bmatrix} 1 & 3 & -9 \\ 2 & 2 & 6 \end{bmatrix}$, 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.

13.7.116 MINVAL (ARRAY, DIM [, MASK]) or MINVAL (ARRAY [, MASK])

1 **Description.** Minimum value(s) of array.

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.

4 **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 $[d_1, d_2, \ldots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \ldots, d_n]$ where $[d_1, d_2, \ldots, d_n]$ is the shape of ARRAY.

5 **Result Value.**

Case (i): The result of 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 LEN (ARRAY), with each character equal to CHAR ($n - 1$, KIND (ARRAY)), where n is the number of characters in the collating sequence for characters with the kind type parameter of ARRAY.

Case (ii): The result of MINVAL (ARRAY, MASK = MASK) has a value equal to that of MINVAL (PACK (ARRAY, MASK)).

Case (iii): The result of MINVAL (ARRAY, DIM = DIM [, MASK = MASK]) has a value equal to that of MINVAL (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

MINVAL (ARRAY $(s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n)$
[, MASK= MASK $(s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n)$]).

6 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.

7 **Examples.**

Case (i): The value of MINVAL ([1, 2, 3]) is 1.

Case (ii): MINVAL (C, MASK = C > 0.0) is the minimum of the positive elements of C.

Case (iii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$, MINVAL (B, DIM = 1) is [1, 3, 5] and MINVAL (B, DIM = 2) is [1, 2].

13.7.117 MOD (A, P)

1 **Description.** Remainder function.

2 **Class.** Elemental function.

3 **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.

4 **Result Characteristics.** Same as A.

5 **Result Value.** The value of the result is A − INT (A/P) * P.

6 **Examples.** MOD (3.0, 2.0) has the value 1.0 (approximately). MOD (8, 5) has the value 3. MOD (−8, 5) has the value −3. MOD (8, −5) has the value 3. MOD (−8, −5) has the value −3.

13.7.118 MODULO (A, P)

1 **Description.** Modulo function.

2 **Class.** Elemental function.

3 **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.

4 **Result Characteristics.** Same as A.

5 **Result Value.**

Case (i): A is of type integer. MODULO (A, P) has the value R such that $A = Q \times P + R$, where Q is an integer, the inequalities $0 \le R < P$ hold if $P > 0$, and $P < R \le 0$ hold if $P < 0$.

Case (ii): A is of type real. The value of the result is A − FLOOR (A / P) * P.

6 **Examples.** MODULO (8, 5) has the value 3. MODULO (−8, 5) has the value 2. MODULO (8, −5) has the value −2. MODULO (−8, −5) has the value −3.

13.7.119 MOVE_ALLOC (FROM, TO)

1 **Description.** Move an allocation.

2 **Class.** Pure subroutine.

3 **Arguments.**

FROM may be of any type, rank, and corank. It shall be allocatable. It is an INTENT (INOUT) argument.

TO shall be type compatible (4.3.2.3) with FROM and have the same rank and corank. It shall be allocatable. 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.

4 The allocation status of TO becomes unallocated if FROM is unallocated on entry to MOVE_ALLOC. Otherwise, TO becomes allocated with dynamic type, type parameters, bounds, cobounds, and value identical to those that FROM had on entry to MOVE_ALLOC.

5 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.

6 The allocation status of FROM becomes unallocated.

7 When a reference to MOVE_ALLOC is executed for which the FROM argument is a coarray, there is an implicit synchronization of all images. On each image, execution of the segment (8.5.2) following the CALL statement is delayed until all other images have executed the same statement the same number of times.

8 **Example.**

9

```
REAL,ALLOCATABLE :: GRID(:),TEMPGRID(:)
...
ALLOCATE(GRID(-N:N))       ! initial allocation of GRID
...
! "reallocation" of GRID to allow intermediate points
ALLOCATE(TEMPGRID(-2*N:2*N)) ! allocate bigger grid
TEMPGRID(::2)=GRID ! distribute values to new locations
CALL MOVE_ALLOC(TO=GRID,FROM=TEMPGRID)
                   ! old grid is deallocated because TO is
                   ! INTENT (OUT), and GRID then "takes over"
                   ! new grid allocation
```

NOTE 13.18

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.

13.7.120 MVBITS (FROM, FROMPOS, LEN, TO, TOPOS)

1 **Description.** Copy a sequence of bits.

2 **Class.** Elemental subroutine.

3 **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 13.3.

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 (12.8.3). 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 13.3.

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).

4 **Example.** If TO has the initial value 6, the value of TO after the statement CALL MVBITS (7, 2, 2, TO, 0) is 5.

13.7.121 NEAREST (X, S)

1 **Description.** Adjacent machine number.

2 **Class.** Elemental function.

3 **Arguments.**

X shall be of type real.

S shall be of type real and not equal to zero.

4 **Result Characteristics.** Same as X.

5 **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 ∞ with the same sign as S.

6 **Example.** NEAREST (3.0, 2.0) has the value $3 + 2^{-22}$ on a machine whose representation is that of the model in Note 13.5.

NOTE 13.19

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.

13.7.122 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.

13.7.123 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.

13.7.124 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.

4 **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 $[d_1, d_2, \ldots, d_{\mathrm{DIM}-1}, d_{\mathrm{DIM}+1}, \ldots, d_n]$, where n is the rank of X and $[d_1, d_2, \ldots, d_n]$ is the shape of X.

5 **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 $(s_1, s_2, \ldots, s_{\mathrm{DIM}-1}, s_{\mathrm{DIM}+1}, \ldots\ s_n)$ of the result is equal to NORM2 (X$(s_1, s_2, \ldots, s_{\mathrm{DIM}-1}, :, s_{\mathrm{DIM}+1}, \ldots\ s_n)$).

6 It is recommended that the processor compute the result without undue overflow or underflow.

7 **Example.** The value of NORM2 ([3.0, 4.0]) is 5.0 (approximately). If X has the value $\begin{bmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \end{bmatrix}$ 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).

13.7.125 NOT (I)

1 **Description.** Bitwise complement.

2 **Class.** Elemental function.

3 **Argument.** I shall be of type integer.

4 **Result Characteristics.** Same as I.

5 **Result Value.** The result has the value obtained by complementing I bit-by-bit according to the following truth table:

I	NOT (I)
1	0
0	1

6 The model for the interpretation of an integer value as a sequence of bits is in 13.3.

7 **Example.** If I is represented by the string of bits 01010101, NOT (I) has the binary value 10101010.

13.7.126 NULL ([MOLD])

1 **Description.** Disassociated pointer or unallocated allocatable entity.

2 **Class.** Transformational function.

3 **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.

4 **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.

5 If MOLD is absent, the characteristics of the result are determined by the entity with which the reference is associated. See Table 13.5. 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.

6 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 13.5: **Characteristics of the result of NULL ( )**

Appearance of NULL ( )	Type, type parameters, and rank of result:
right side of a pointer assignment	pointer on the left side
initialization for an object in a declaration	the object
default initialization for a component	the component
in a structure constructor	the corresponding component
as an actual argument	the corresponding dummy argument
in a DATA statement	the corresponding pointer object

7 **Result.** The result is a disassociated pointer or an unallocated allocatable entity.

8 **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:

```
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
```

13.7.127 NUM_IMAGES ( )

1 **Description.** Number of images.

2 **Class.** Transformational function.

3 **Argument.** None.

4 **Result Characteristics.** Default integer scalar.

5 **Result Value.** The number of images.

6 **Example.** The following code uses image 1 to read data and broadcast it to other images.

7

```
REAL :: P[*]
IF (THIS_IMAGE()==1) THEN
   READ (6,*) P
   DO I = 2, NUM_IMAGES()
      P[I] = P
   END DO
```

```
        END IF
        SYNC ALL
```

13.7.128 OUT_OF_RANGE (X, MOLD, [, ROUND])

1 **Description.** Whether a value cannot be converted safely.

2 **Class.** Elemental function.

3 **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.

4 **Result Characteristics.** Default logical scalar.

5 **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.

6 **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.

13.7.129 PACK (ARRAY, MASK [, VECTOR])

1 **Description.** Array packed into a vector.

2 **Class.** Transformational function.

3 **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.

4 **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.

5 **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 $i = t + 1, \ldots, n$.

6 **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].

13.7.130 PARITY (MASK) or PARITY (MASK, DIM)

1 **Description.** Array reduced by .NEQV. operation.

2 **Class.** Transformational function.

3 **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.

4 **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.

5 **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 $(s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n)$).

6 **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].

13.7.131 POPCNT (I)

1 **Description.** Number of one bits.

2 **Class.** Elemental function.

3 **Argument.** I shall be of type integer.

4 **Result Characteristics.** Default integer.

5 **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 13.3.

6 **Examples.** POPCNT ([1, 2, 3, 4, 5, 6]) has the value [1, 1, 2, 1, 2, 2].

13.7.132 POPPAR (I)

1 **Description.** Parity expressed as 0 or 1.

2 **Class.** Elemental function.

3 **Argument.** I shall be of type integer.

4 **Result Characteristics.** Default integer.

5 **Result Value.** POPPAR (I) has the value 1 if POPCNT (I) is odd, and 0 if POPCNT (I) is even.

6 **Examples.** POPPAR ([1, 2, 3, 4, 5, 6]) has the value [1, 1, 0, 1, 0, 0].

13.7.133 PRECISION (X)

1 **Description.** Decimal precision of a real model.

2 **Class.** Inquiry function.

3 **Argument.** X shall be of type real or complex. It may be a scalar or an array.

4 **Result Characteristics.** Default integer scalar.

5 **Result Value.** The result has the value INT $((p - 1)$ * LOG10 $(b)) + k$, where b and p are as defined in 13.4 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.

6 **Example.** PRECISION (X) has the value INT (23 * LOG10 (2.)) = INT (6.92...) = 6 for real X whose model is as in Note 13.5.

13.7.134 PRESENT (A)

1 **Description.** Presence of optional argument.

2 **Class.** Inquiry function.

3 **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.

4 **Result Characteristics.** Default logical scalar.

5 **Result Value.** The result has the value true if A is present (12.5.2.12) and otherwise has the value false.

13.7.135 PRODUCT (ARRAY, DIM [, MASK]) or PRODUCT (ARRAY [, MASK])

1 **Description.** Array reduced by multiplication.

2 **Class.** Transformational function.

3 **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 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 (s_1, s_2, ..., $s_{\text{DIM}-1}$, $s_{\text{DIM}+1}$, ..., s_n) of PRODUCT (ARRAY, DIM = DIM [, MASK = MASK]) is equal to

PRODUCT (ARRAY (s_1, s_2, ..., $s_{\text{DIM}-1}$, :, $s_{\text{DIM}+1}$, ..., s_n) [, MASK = MASK (s_1, s_2, ..., $s_{\text{DIM}-1}$, :, $s_{\text{DIM}+1}$, ..., 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].

13.7.136 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 13.4 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 13.5.

13.7.137 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 set by calls to RANDOM_INIT in other images of the program. 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 other images use distinct seeds:

```
CALL RANDOM_INIT (REPEATABLE=.FALSE., IMAGE_DISTINCT=.TRUE.)
```

Unresolved Technical Issue 010

RANDOM_INIT example text claim is not consistent with it being processor-dependent whether the generator is common or independent.

Unresolved Technical Issue 010 (cont.)

Furthermore, it does not even follow from the above normative text (though that is not itself entirely without problems!). Perhaps the normative text should clearly state the differences in what happens between the common (per-program) and independent (per-image) generator cases.

13.7.138 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.**

```
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
```

13.7.139 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

```
   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

```
   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.

6 **Examples.**

```
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
```

13.7.140 RANGE (X)

1 **Description.** Decimal exponent range of a numeric model (13.4).

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 13.5, because in this case HUGE (X) $= (1 - 2^{-24}) \times 2^{127}$ and TINY (X) $= 2^{-127}$.

13.7.141 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.

13.7.142 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 *boz-literal-constant* and KIND is present, the kind type parameter is that specified by the value of KIND. If A is a *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 *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 13.3.3. 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.

13.7.143 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 pure function with two arguments of the same type and type parameters as ARRAY. Its result shall have the same type and type parameters as ARRAY. The arguments and result shall not be 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 conformable with ARRAY.

IDENTITY (optional) shall be scalar with the same type and type parameters as ARRAY.

ORDERED (optional) shall be a logical scalar.

4 **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 $[d_1, d_2, \ldots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \ldots, d_n]$ 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 is 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 $(s_1, s_2, \ldots, s_{\mathrm{DIM}-1}, s_{\mathrm{DIM}+1}, \ldots, s_n)$ of REDUCE (ARRAY, DIM = DIM [, MASK = MASK, IDENTITY = IDENTITY]) is equal to

REDUCE (ARRAY $(s_1, s_2, \ldots, s_{\mathrm{DIM}-1}, :, s_{\mathrm{DIM}+1}, \ldots, s_n)$,
OPERATION = OPERATION,
DIM=1
[, MASK = MASK $(s_1, s_2, \ldots, s_{\mathrm{DIM}-1}, :, s_{\mathrm{DIM}+1}, \ldots, s_n)$,
IDENTITY = IDENTITY,
ORDERED = ORDERED]).

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 13.20

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.

13.7.144 REPEAT (STRING, NCOPIES)

1 **Description.** Repetitive string concatenation.

2 **Class.** Transformational function.

3 **Arguments.**

STRING shall be a character scalar.

NCOPIES shall be an integer scalar. Its value shall not be negative.

4 **Result Characteristics.** Character scalar of length NCOPIES times that of STRING, with the same kind type parameter as STRING.

5 **Result Value.** The value of the result is the concatenation of NCOPIES copies of STRING.

6 **Examples.** REPEAT ('H', 2) has the value HH. REPEAT ('XYZ', 0) has the value of a zero-length string.

13.7.145 RESHAPE (SOURCE, SHAPE [, PAD, ORDER])

1 **Description.** Arbitrary shape array construction.

2 **Class.** Transformational function.

3 **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 $(1, 2, \ldots, n)$.

4 **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.

5 **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.

6 **Examples.** RESHAPE ([1, 2, 3, 4, 5, 6], [2, 3]) has the value $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$.

RESHAPE ([1, 2, 3, 4, 5, 6], [2, 4], [0, 0], [2, 1]) has the value $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 0 & 0 \end{bmatrix}$.

13.7.146 RRSPACING (X)

1 **Description.** Reciprocal of relative 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.** The result has the value $|\mathrm{Y} \times b^{-e}| \times b^p$ = ABS (FRACTION (Y)) * RADIX (X) / EPSILON (X), where b, e, and p are as defined in 13.4 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.

6 **Example.** RRSPACING (-3.0) has the value 0.75×2^{24} for reals whose model is as in Note 13.5.

13.7.147 SAME_TYPE_AS (A, B)

1 **Description.** Dynamic type equality test.

2 **Class.** Inquiry function.

3 **Arguments.**

A shall be an object of extensible declared type or unlimited polymorphic. If it is a 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 pointer, it shall not have an undefined association status.

4 **Result Characteristics.** Default logical scalar.

5 **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 13.21

The dynamic type of a disassociated pointer or unallocated allocatable variable is its declared type. An unlimited polymorphic entity has no declared type.

13.7.148 SCALE (X, I)

1 **Description.** Real number scaled by radix power.

2 **Class.** Elemental function.

3 **Arguments.**

X	shall be of type real.
I	shall be of type integer.

4 **Result Characteristics.** Same as X.

5 **Result Value.** The result has the value $X \times b^{I}$, where b is defined in 13.4 for model numbers representing values of X, provided this result is representable; if not, the result is processor dependent.

6 **Example.** SCALE (3.0, 2) has the value 12.0 for reals whose model is as in Note 13.5.

13.7.149 SCAN (STRING, SET [, BACK, KIND])

1 **Description.** Character set 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 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.

6 **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.

13.7.150 SELECTED_CHAR_KIND (NAME)

1 **Description.** Character kind selection.

2 **Class.** Transformational function.

3 **Argument.** NAME shall be default character scalar.

4 **Result Characteristics.** Default integer scalar.

5 **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.

6 **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.

7
```
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
```

13.7.151 SELECTED_INT_KIND (R)

1 **Description.** Integer kind selection.

2 **Class.** Transformational function.

3 **Argument.** R shall be an integer scalar.

4 **Result Characteristics.** Default integer scalar.

5 **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 $-10^{\mathrm{R}} < n < 10^{\mathrm{R}}$, 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.

6 **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.

13.7.152 SELECTED_REAL_KIND ([P, R, RADIX])

1 **Description.** Real kind selection.

2 **Class.** Transformational function.

3 **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.

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 $b = 16$, $p = 6$, $e_{\min} = -64$, and $e_{\max} = 63$ and does not have a less precise approximation method.

13.7.153 SET_EXPONENT (X, I)

1 **Description.** Real value with specified exponent.

2 **Class.** Elemental function.

3 **Arguments.**

X shall be of type real.

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 $\mathrm{X} \times b^{\mathrm{I}-e}$, where b and e are as defined in 13.4 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 13.5.

13.7.154 SHAPE (SOURCE [, KIND])

1 **Description.** Shape of an array or a scalar.

2 **Class.** Inquiry function.

3 **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.

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 rank of SOURCE.

5 **Result Value.** The result has a value whose i^{th} element is equal to the extent of 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.

6 **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.

13.7.155 SHIFTA (I, SHIFT)

1 **Description.** Right shift with fill.

2 **Class.** Elemental function.

3 **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).

4 **Result Characteristics.** Same as I.

5 **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.

6 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 13.3.

7 **Example.** SHIFTA (IBSET (0, BIT_SIZE (0) − 1), 2) is equal to SHIFTL (7, BIT_SIZE (0) − 3).

13.7.156 SHIFTL (I, SHIFT)

1 **Description.** Left shift.

2 **Class.** Elemental function.

3 **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).

4 **Result Characteristics.** Same as I.

5 **Result Value.** The value of the result is ISHFT (I, SHIFT).

6 **Examples.** SHIFTL (3, 1) has the value 6.

13.7.157 SHIFTR (I, SHIFT)

1 **Description.** Right shift.

2 **Class.** Elemental function.

3 **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).

4 **Result Characteristics.** Same as I.

5 **Result Value.** The value of the result is ISHFT (I, −SHIFT).

6 **Examples.** SHIFTR (3, 1) has the value 1.

13.7.158 SIGN (A, B)

1 **Description.** Magnitude of A with the sign of B.

2 **Class.** Elemental function.

3 **Arguments.**

A	shall be of type integer or real.
B	shall be of the same type and kind type parameter as A.

4 **Result Characteristics.** Same as A.

5 **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 not distinguish between positive and negative real zero, or if B is positive real zero, the value of the result is |A|;
- if B is negative real zero, the value of the result is -|A|.

6 **Example.** SIGN (−3.0, 2.0) has the value 3.0.

13.7.159 SIN (X)

1 **Description.** 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 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.

6 **Example.** SIN (1.0) has the value 0.84147098 (approximately).

13.7.160 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).

13.7.161 SIZE (ARRAY [, DIM, KIND])

1 **Description.** Size of an array or one extent.

2 **Class.** Inquiry function.

3 **Arguments.**

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.

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.

KIND (optional) shall be a scalar integer constant expression.

4 **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.

5 **Result Value.** If DIM is present, the result has a value equal to the extent of 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.

6 If DIM is absent and ARRAY is assumed-rank, the result has a value equal to PRODUCT(SHAPE(ARRAY, KIND)). Otherwise, the result has a value equal to the total number of elements of ARRAY.

7 **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 13.22

If ARRAY is assumed-rank and has rank zero, DIM cannot be present since it cannot satisfy the requirement $1 \leq \text{DIM} \leq 0$.

13.7.162 SPACING (X)

1 **Description.** Spacing of model numbers (13.4).

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 $b^{\max(e-p, e_{\min}-1)}$, where b, e, and p are as defined in 13.4 for 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 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.

6 **Example.** SPACING (3.0) has the value 2^{-22} for reals whose model is as in Note 13.5.

13.7.163 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 $1 \leq \text{DIM} \leq n+1$, 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 (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 $[d_1, d_2, \ldots, d_{\text{DIM}-1}$, 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 $\begin{bmatrix} 2 & 3 & 4 \\ 2 & 3 & 4 \\ 2 & 3 & 4 \end{bmatrix}$ if NC has the value 3 and is a zero-sized array if NC has the value 0.

13.7.164 SQRT (X)

1 **Description.** Square root.

2 **Class.** Elemental function.

3 **Argument.** X shall be of type real or complex. Unless X is complex, 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).

13.7.165 STORAGE_SIZE (A [, KIND])

1 **Description.** Storage size in bits.

2 **Class.** Inquiry function.

3 **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.

4 **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.

5 **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 (16.5.3) applies, the result is consistent with the named constants defined in the intrinsic module ISO_FORTRAN_ENV.

NOTE 13.23

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 13.24

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.

6 **Example.** STORAGE_SIZE (1.0) has the same value as the named constant NUMERIC_STORAGE_SIZE in the intrinsic module ISO_FORTRAN_ENV.

13.7.166 SUM (ARRAY, DIM [, MASK]) or SUM (ARRAY [, MASK])

1 **Description.** Array reduced by addition.

2 **Class.** Transformational function.

3 **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

SUM (ARRAY $(s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n)$ [, MASK= MASK $(s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n)$]).

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 $[9,\ 12]$.

13.7.167 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 −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

```
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 $M = 24 \times 3600 \times 18.20648193 - 1 = 1573039$.

13.7.168 TAN (X)

1 **Description.** Tangent 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 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.

6 **Example.** TAN (1.0) has the value 1.5574077 (approximately).

13.7.169 TANH (X)

1 **Description.** Hyperbolic tangent 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 tanh(X). If X is of type complex its imaginary part is regarded as a value in radians.

6 **Example.** TANH (1.0) has the value 0.76159416 (approximately).

13.7.170 THIS_IMAGE ( ) or THIS_IMAGE (COARRAY) or THIS_IMAGE (COARRAY, DIM)

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.

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.

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 ( ) is a scalar with a value equal to the index of the invoking image.

Case (ii): The result of THIS_IMAGE (COARRAY) is the sequence of cosubscript values for COARRAY that would specify the invoking image.

Case (iii): The result of THIS_IMAGE (COARRAY, DIM) is the value of cosubscript DIM in the sequence of cosubscript values for COARRAY that would specify the invoking image.

6 **Examples.** If A is declared by the statement

```
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].

7 The following code uses image 1 to read data. The other images then copy the data.

```
IF (THIS_IMAGE()==1) READ (*,*) P
SYNC ALL
P = P[1]
```

13.7.171 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 $b^{e_{\text{min}}-1}$ where b and e_{min} are as defined in 13.4 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 13.5.

13.7.172 TRAILZ (I)

1 **Description.** Number of trailing zero bits.

2 **Class.** Elemental function.

3 **Argument.** I shall be of type integer.

4 **Result Characteristics.** Default integer.

5 **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 13.3.

6 **Examples.** TRAILZ (8) has the value 3.

13.7.173 TRANSFER (SOURCE, MOLD [, SIZE])

1 **Description.** Transfer physical representation.

2 **Class.** Transformational function.

3 **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.

SIZE (optional) shall be an integer scalar. The corresponding actual argument shall not be an optional dummy argument.

4 **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.

5 **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 result is the leading part of SOURCE. 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.

6 **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).

13.7.174 TRANSPOSE (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 $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$, then TRANSPOSE (A) has the value $\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$.

13.7.175 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.** TRIM (' A B ') has the value ' A B'.

13.7.176 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.

6 **Examples.** If A is declared by the statement

```
REAL A (2:3, 7:10)
```

then UBOUND (A) is [3, 10] and UBOUND (A, DIM = 2) is 10.

NOTE 13.25

If ARRAY is assumed-rank and has rank zero, DIM cannot be present since it cannot satisfy the requirement $1 \leq \text{DIM} \leq 0$.

13.7.177 UCOBOUND (COARRAY [, DIM, KIND])

1 **Description.** Upper cobound(s) of a coarray.

2 **Class.** Inquiry function.

3 **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.

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 corank of COARRAY.

5 **Result Value.** The final upper cobound is the final cosubscript in the cosubscript list for the coarray that selects the image with index NUM_IMAGES( ).

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.

6 **Examples.** If NUM_IMAGES( ) has the value 30 and A is allocated by the statement

```
ALLOCATE (A [2:3, 0:7, *])
```

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.

13.7.178 UNPACK (VECTOR, MASK, FIELD)

1 **Description.** Vector unpacked into an array.

2 **Class.** Transformational function.

3 **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 $\begin{bmatrix} 0 & 2 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 3 \end{bmatrix}$.

13.7.179 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.

13.8 Standard modules

13.8.1 General

1 This part of ISO/IEC 1539 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 14. The intrinsic module ISO_C_BINDING is described in Clause 15. The module procedures described in 13.8.2 are pure.

NOTE 13.26

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 part of ISO/IEC 1539.

13.8.2 The ISO_FORTRAN_ENV intrinsic module

13.8.2.1 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 type, and procedures described in subclause 13.8.2. In the detailed descriptions below, procedure names are generic and not specific.

13.8.2.2 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.

13.8.2.3 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.

13.8.2.4 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.

13.8.2.5 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 (16.5.3.2).

13.8.2.6 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'.

13.8.2.7 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 13.27

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.

13.8.2.8 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 (9.5). This unit may be the same as OUTPUT_UNIT. The value shall not be −1.

13.8.2.9 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 (9.3.5).

13.8.2.10 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 (9.6.4.3). The value shall not be −1.

13.8.2.11 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.

13.8.2.12 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.

13.8.2.13 IOSTAT_END

1 The value of the default integer scalar constant IOSTAT_END is assigned to the variable specified in an IOSTAT= specifier (9.11.5) if an end-of-file condition occurs during execution of an input statement and no error condition occurs. This value shall be negative.

13.8.2.14 IOSTAT_EOR

1 The value of the default integer scalar constant IOSTAT_EOR is assigned to the variable specified in an IOSTAT= specifier (9.11.5) 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.

13.8.2.15 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 (9.10) if a *file-unit-number* identifies an internal unit in that statement.

NOTE 13.28

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 (9.6.4.8.2) for an internal unit.

13.8.2.16 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 initial 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 (8.5.6).

C1302 A named variable of type LOCK_TYPE shall be a coarray. A named variable with a noncoarray subcomponent of type LOCK_TYPE shall be a coarray.

C1303 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).

C1304 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 13.29

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.

13.8.2.17 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.

13.8.2.18 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 (16.5.3.2).

13.8.2.19 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 (9.6.4.3). The value shall not be −1.

13.8.2.20 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.

13.8.2.21 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.

13.8.2.22 STAT_LOCKED

1 The value of the default integer scalar constant STAT_LOCKED is assigned to the variable specified in a STAT= specifier (8.5.7) of a LOCK statement if the lock variable is locked by the executing image.

13.8.2.23 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 (8.5.7) of an UNLOCK statement if the lock variable is locked by another image.

13.8.2.24 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 (6.7.4, 8.5.7) if execution of the statement with that specifier or argument requires synchronization with an image that has initiated termination of execution. This value shall be positive and different from the value of IOSTAT_INQUIRE_INTERNAL_UNIT.

13.8.2.25 STAT_UNLOCKED

1 The value of the default integer scalar constant STAT_UNLOCKED is assigned to the variable specified in a STAT= specifier (8.5.7) of an UNLOCK statement if the lock variable is unlocked.

(Blank page)

14 Exceptions and IEEE arithmetic

14.1 General

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 part of ISO/IEC 1539 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 14.1

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 (14.11.45); 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 14.2

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) (14.11.42) 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 which IEEE_SUPPORT_-

* 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 part of ISO/IEC 1539.

DATATYPE (X) is true and RADIX (X) is equal to ten. Note that the effect of IEEE_DENORMAL is the same as that of IEEE_SUBNORMAL.

5 Execution might be slowed on some processors by the support of some features. If IEEE_EXCEPTIONS or IEEE_ARITHMETIC is accessed but IEEE_FEATURES is not accessed, the supported subset of features is processor dependent. The processor's fullest support is provided when all of IEEE_FEATURES is accessed as in

```
USE, INTRINSIC :: IEEE_ARITHMETIC; USE, INTRINSIC :: IEEE_FEATURES
```

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.

14.2 Derived types and constants defined in the modules

1 The modules IEEE_EXCEPTIONS, IEEE_ARITHMETIC, and IEEE_FEATURES define five derived types, whose components are all private. No direct component of any of these types is allocatable or a pointer.

2 The module IEEE_EXCEPTIONS defines the following types.

- IEEE_FLAG_TYPE is for identifying a particular exception flag. Its only possible values are those of named constants defined in the module: IEEE_INVALID, IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_UNDERFLOW, and IEEE_INEXACT. The module also defines the array named constants IEEE_USUAL = [IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_INVALID] and IEEE_ALL = [IEEE_USUAL, IEEE_UNDERFLOW, IEEE_INEXACT].
- IEEE_MODES_TYPE is for representing the floating-point modes.
- IEEE_STATUS_TYPE is for representing the floating-point status.

3 The module IEEE_ARITHMETIC defines the following.

- The type IEEE_CLASS_TYPE, for identifying a class of floating-point values. Its only possible values are those of named constants defined in the module: IEEE_SIGNALING_NAN, IEEE_QUIET_NAN, IEEE_NEGATIVE_INF, IEEE_NEGATIVE_NORMAL, IEEE_NEGATIVE_DENORMAL, IEEE_NEGATIVE_ZERO, IEEE_POSITIVE_ZERO, IEEE_POSITIVE_SUBNORMAL, IEEE_POSITIVE_NORMAL, IEEE_POSITIVE_INF, and IEEE_OTHER_VALUE. The named constants IEEE_NEGATIVE_DENORMAL and IEEE_POSITIVE_DENORMAL are defined with the same value as IEEE_NEGATIVE_SUBNORMAL and IEEE_POSITIVE_SUBNORMAL respectively.
- The type IEEE_ROUND_TYPE, for identifying a particular rounding mode. Its only possible values are those of named constants defined in the module: IEEE_NEAREST, IEEE_TO_ZERO, IEEE_UP, IEEE_DOWN, and IEEE_AWAY for the rounding modes specified in ISO/IEC/IEEE 60559:2011, and IEEE_OTHER for any other mode.
- The elemental operator == for two values of one of these types to return true if the values are the same and false otherwise.
- The elemental operator /= for two values of one of these types to return true if the values differ and false otherwise.

4 The module IEEE_FEATURES defines the type 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: IEEE_DATATYPE, IEEE_DENORMAL, IEEE_DIVIDE, IEEE_HALTING, IEEE_INEXACT_FLAG, IEEE_INF, IEEE_INVALID_FLAG, IEEE_NAN, IEEE_ROUNDING, IEEE_SQRT, IEEE_SUBNORMAL, and IEEE_UNDERFLOW_FLAG.

14.3 The exceptions

1 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. In a numeric relational operation x_1 *rel-op* x_2, if $x_1 + x_2$ is of type real and IEEE_SUPPORT_NAN ($x_1 + x_2$) is true, IEEE_INVALID shall signal as specified by ISO/IEC/IEEE 60559:2011 for the compareSignaling{relation} operations; that is, if x_1 and x_2 are unordered.
- 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_FLAG 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.

NOTE 14.3

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 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.

7 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

```
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

```
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 .OR. X/Y>0.00001
```

is permitted to cause the signaling of IEEE_DIVIDE_BY_ZERO.

8 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.

14.4 The rounding modes

1 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.

2 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.

3 The subroutine IEEE_GET_ROUNDING_MODE can be used to get the rounding modes. The initial rounding modes are processor dependent.

4 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 (9.5.6.16, 9.6.2.13, 10.7.2.3.8) is as specified in ISO/IEC/IEEE 60559:2011.

5 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 14.4

Within a program, all literal constants that have the same form have the same value (4.1.4). Therefore, the value of a literal constant is not affected by the rounding modes.

14.5 Underflow mode

1 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.

2 Control over the underflow mode is exercised by invocation of IEEE_SET_UNDERFLOW_MODE. The subroutine 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.

3 The underflow mode affects only floating-point calculations whose type is that of an X for which IEEE_SUPPORT_UNDERFLOW_CONTROL returns true.

14.6 Halting

1 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_SUPPORT_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.

14.7 The floating-point modes and status

1 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 14.5

Each image has its own floating-point status (2.3.4).

NOTE 14.6

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 14.7

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.

14.8 Exceptional values

1 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.

2 A value that does not fall into the above classes is called a normal number.

3 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.

14.9 IEEE arithmetic

1 The inquiry function IEEE_SUPPORT_DATATYPE can be used to inquire whether IEEE arithmetic is supported 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 infinities. 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 operand 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 division 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 part of ISO/IEC 1539 for a particular kind of real.

14.10 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 14.10 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 neither an inquiry function nor elemental.

3 In the Class column of Tables 14.1 and 14.2,

- 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 14.1: **IEEE_ARITHMETIC module procedure summary**

Procedure	Arguments	Class	Description
IEEE_CLASS	(X)	E	Classify number.
IEEE_COPY_SIGN	(X, Y)	E	Copy sign.
IEEE_FMA	(A, B, C)	E	Fused multiply-add operation.
IEEE_GET_ROUNDING_MODE	(ROUND_VALUE [, RADIX])	S	Get rounding mode.
IEEE_GET_UNDERFLOW_MODE	(GRADUAL)	S	Get underflow mode.
IEEE_INT	(A, ROUND [, KIND])	E	Conversion to integer type.
IEEE_IS_FINITE	(X)	E	Whether a value is finite.
IEEE_IS_NAN	(X)	E	Whether a value is an IEEE NaN.
IEEE_IS_NEGATIVE	(X)	E	Whether a value is negative.
IEEE_IS_NORMAL	(X)	E	Whether a value is a normal number.
IEEE_LOGB	(X)	E	Exponent.
IEEE_MAX_NUM	(X, Y)	E	Maximum numeric value.
IEEE_MAX_NUM_MAG	(X, Y)	E	Maximum magnitude numeric value.
IEEE_MIN_NUM	(X, Y)	E	Minimum numeric value.
IEEE_MIN_NUM_MAG	(X, Y)	E	Minimum magnitude numeric value.
IEEE_NEXT_AFTER	(X, Y)	E	Adjacent machine number.

Table 14.1: **IEEE_ARITHMETIC module procedure summary** (cont.)

Procedure	Arguments	Class	Description
IEEE_NEXT_DOWN	(X)	E	Adjacent lower machine number.
IEEE_NEXT_UP	(X)	E	Adjacent higher machine number.
IEEE_QUIET_EQ	(A, B)	E	Quiet compares equal.
IEEE_QUIET_GE	(A, B)	E	Quiet compares greater than or equal.
IEEE_QUIET_GT	(A, B)	E	Quiet compares greater than.
IEEE_QUIET_LE	(A, B)	E	Quiet compares less than or equal.
IEEE_QUIET_LT	(A, B)	E	Quiet compares less than.
IEEE_QUIET_NE	(A, B)	E	Quiet compares not equal.
IEEE_REAL	(A [, KIND])	E	Conversion to real type.
IEEE_REM	(X, Y)	E	Exact remainder.
IEEE_RINT	(X)	E	Round to integer.
IEEE_SCALB	(X, I)	E	$X \times 2^I$.
IEEE_SELECTED_REAL_KIND	([P, R, RADIX])	T	IEEE kind type parameter value.
IEEE_SET_ROUNDING_MODE	(ROUND_VALUE [, RADIX])	S	Set rounding mode.
IEEE_SET_UNDERFLOW_MODE	(GRADUAL)	S	Set underflow mode.
IEEE_SIGNBIT	(X)	E	Test sign bit.
IEEE_SUPPORT_DATATYPE	([X])	I	Query IEEE arithmetic support.
IEEE_SUPPORT_DENORMAL	([X])	I	Query subnormal number support.
IEEE_SUPPORT_DIVIDE	([X])	I	Query IEEE division support.
IEEE_SUPPORT_INF	([X])	I	Query IEEE infinity support.
IEEE_SUPPORT_IO	([X])	I	Query IEEE formatting support.
IEEE_SUPPORT_NAN	([X])	I	Query IEEE NaN support.
IEEE_SUPPORT_ROUNDING	(ROUND_VALUE [, X])	I	Query IEEE rounding support.
IEEE_SUPPORT_SQRT	([X])	I	Query IEEE square root support.
IEEE_SUPPORT_SUBNORMAL	([X])	I	Query subnormal number support.
IEEE_SUPPORT_STANDARD	([X])	I	Query IEEE standard support.
IEEE_SUPPORT_UNDERFLOW_CONTROL	([X])	I	Query underflow control support.
IEEE_UNORDERED	(X, Y)	E	Whether two values are unordered.
IEEE_VALUE	(X, CLASS)	E	Return number in a class.

Table 14.2: **IEEE_EXCEPTIONS module procedure summary**

Procedure	Arguments	Class	Description
IEEE_GET_FLAG	(FLAG, FLAG_VALUE)	ES	Get an exception flag.
IEEE_GET_HALTING_MODE	(FLAG, HALTING)	ES	Get a halting mode.
IEEE_GET_MODES	(MODES)	S	Get floating-point modes.
IEEE_GET_STATUS	(STATUS_VALUE)	S	Get floating-point state.
IEEE_SET_FLAG	(FLAG, FLAG_VALUE)	PS	Set an exception flag.
IEEE_SET_HALTING_MODE	(FLAG, HALTING)	PS	Set a halting mode.
IEEE_SET_MODES	(MODES)	S	Set floating-point modes.
IEEE_SET_STATUS	(STATUS_VALUE)	S	Restore floating-point state.
IEEE_SUPPORT_FLAG	(FLAG [, X])	I	Query exception support.
IEEE_SUPPORT_HALTING	(FLAG)	I	Query halting mode support.

4 In the intrinsic module IEEE_ARITHMETIC, the elemental functions listed are provided for all reals X and Y.

14.11 Specifications of the procedures

14.11.1 General

1 In the detailed descriptions in 14.11, procedure names are generic and are not specific. All the functions are pure. 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 14.8

A program can avoid violating those requirements by using IF constructs to check whether particular features are supported. For example,

```
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.

14.11.2 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 a 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 14.9

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.

14.11.3 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.

14.11.4 IEEE_FMA (A, B, C)

1 **Description.** Fused multiply-add operation.

2 **Class.** Elemental function.

3 **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.

4 **Restriction.** IEEE_FMA (A, B, C) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the value false.

5 **Result Characteristics.** Same as A.

6 **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 $(A \times B) + C$ 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.

7 **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.

14.11.5 IEEE_GET_FLAG (FLAG, FLAG_VALUE)

1 **Description.** Get an exception flag.

2 **Class.** Elemental subroutine.

3 **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.

4 **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.

14.11.6 IEEE_GET_HALTING_MODE (FLAG, HALTING)

1 **Description.** Get a halting mode.

2 **Class.** Elemental subroutine.

3 **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.

4 **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
```

14.11.7 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.
```

14.11.8 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
```

14.11.9 IEEE_GET_STATUS (STATUS_VALUE)

1 **Description.** Get floating-point state.

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
```

14.11.10 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.

14.11.11 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.

7 **Example.** The value of IEEE_INT (12.5, IEEE_UP) is 13; IEEE_INEXACT will be signaled if the processor provides the convertToIntegerExact operation.

14.11.12 IEEE_IS_FINITE (X)

1 **Description.** Whether a value is finite.

2 **Class.** Elemental function.

3 **Argument.** X shall be of type real.

4 **Restriction.** IEEE_IS_FINITE (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 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.

7 **Example.** IEEE_IS_FINITE (1.0) has the value true.

14.11.13 IEEE_IS_NAN (X)

1 **Description.** Whether a value is an IEEE NaN.

2 **Class.** Elemental function.

3 **Argument.** X shall be of type real.

4 **Restriction.** IEEE_IS_NAN (X) shall not be invoked if IEEE_SUPPORT_NAN (X) has the value false.

5 **Result Characteristics.** Default logical.

6 **Result Value.** The result has the value true if the value of X is an IEEE NaN; otherwise, it has the value false.

7 **Example.** IEEE_IS_NAN (SQRT (−1.0)) has the value true if IEEE_SUPPORT_SQRT (1.0) has the value true.

14.11.14 IEEE_IS_NEGATIVE (X)

1 **Description.** Whether a value is negative.

2 **Class.** Elemental function.

3 **Argument.** X shall be of type real.

4 **Restriction.** IEEE_IS_NEGATIVE (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 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.

7 **Example.** IEEE_IS_NEGATIVE (0.0) has the value false.

14.11.15 IEEE_IS_NORMAL (X)

1 **Description.** Whether a value is a normal number.

2 **Class.** Elemental function.

3 **Argument.** X shall be of type real.

4 **Restriction.** IEEE_IS_NORMAL (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 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.

7 **Example.** IEEE_IS_NORMAL (SQRT (-1.0) has the value false if IEEE_SUPPORT_SQRT (1.0) has the value true.

14.11.16 IEEE_LOGB (X)

1 **Description.** Exponent.

2 **Class.** Elemental function.

3 **Argument.** X shall be of type real.

4 **Restriction.** IEEE_LOGB (X) shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the value false.

5 **Result Characteristics.** Same as X.

6 **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.

7 **Example.** IEEE_LOGB (−1.1) has the value 0.0.

14.11.17 IEEE_MAX_NUM (X, Y)

1 **Description.** Maximum numeric value.

2 **Class.** Elemental function.

3 **Arguments.**

X shall be of type real.

Y shall be of the same type and kind type parameter as X.

4 **Restriction.** IEEE_MAX_NUM shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the value false.

5 **Result Characteristics.** Same as X.

6 **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.

7 Except when X or Y is a signaling NaN, no exception is signaled.

8 **Example.** The value of IEEE_MAX_NUM (1.5, IEEE_VALUE (IEEE_QUIET_NAN)) is 1.5.

14.11.18 IEEE_MAX_NUM_MAG (X, Y)

1 **Description.** Maximum magnitude numeric value.

2 **Class.** Elemental function.

3 **Arguments.**

X shall be of type real.

Y shall be of the same type and kind type parameter as X.

4 **Restriction.** IEEE_MAX_NUM_MAG shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the value false.

5 **Result Characteristics.** Same as X.

6 **Result Value.** The result has the value specified for the maxNumMag operation in ISO/IEC/IEEE 60559:2011; that is,

- if ABS (X) < ABS (Y) the result has the value of Y;
- if ABS (Y) < ABS (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.

7 Except when X or Y is a signaling NaN, no exception is signaled.

8 **Example.** The value of IEEE_MAX_NUM_MAG (1.5, −2.5) is −2.5.

14.11.19 IEEE_MIN_NUM (X, Y)

1 **Description.** Minimum numeric value.

2 **Class.** Elemental function.

3 **Arguments.**

X	shall be of type real.
Y	shall be of the same type and kind type parameter as X.

4 **Restriction.** IEEE_MIN_NUM shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the value false.

5 **Result Characteristics.** Same as X.

6 **Result Value.**

7 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.

8 Except when X or Y is a signaling NaN, no exception is signaled.

9 **Example.** The value of IEEE_MIN_NUM (1.5, IEEE_VALUE (IEEE_QUIET_NAN)) is 1.5.

14.11.20 IEEE_MIN_NUM_MAG (X, Y)

1 **Description.** Minimum magnitude numeric value.

2 **Class.** Elemental function.

3 **Arguments.**

X	shall be of type real.
Y	shall be of the same type and kind type parameter as X.

4 **Restriction.** IEEE_MIN_NUM_MAG shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the value false.

5 **Result Characteristics.** Same as X.

6 **Result Value.** The result has the value specified for the minNumMag operation in ISO/IEC/IEEE 60559:2011; that is,

- if ABS (X) < ABS (Y) the result has the value of X;
- if ABS (Y) < 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.

7 Except when X or Y is a signaling NaN, no exception is signaled.

8 **Example.** The value of IEEE_MIN_NUM_MAG (1.5, −2.5) is 1.5.

14.11.21 IEEE_NEXT_AFTER (X, Y)

1 **Description.** Adjacent machine number.

2 **Class.** Elemental function.

3 **Arguments.** The arguments shall be of type real.

4 **Restriction.** IEEE_NEXT_AFTER (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.**

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.

7 **Example.** The value of IEEE_NEXT_AFTER (1.0, 2.0) is 1.0 + EPSILON (X).

14.11.22 IEEE_NEXT_DOWN (X)

1 **Description.** Adjacent lower machine number.

2 **Class.** Elemental function.

3 **Argument.** X shall be of type real.

4 **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.

5 **Result Characteristics.** Same as X.

6 **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 $-\infty$ the result has the value $-\infty$, 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.

7 **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.

14.11.23 IEEE_NEXT_UP (X)

1 **Description.** Adjacent higher machine number.

2 **Class.** Elemental function.

3 **Argument.** X shall be of type real.

4 **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.

5 **Result Characteristics.** Same as X.

6 **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 $+\infty$ the result has the value $+\infty$, 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.

7 **Example.** If IEEE_SUPPORT_INF (X) is true, the value of IEEE_NEXT_UP (HUGE (X)) is $+\infty$.

14.11.24 IEEE_QUIET_EQ (A, B)

1 **Description.** Quiet compares equal.

2 **Class.** Elemental function.

3 **Arguments.**

A shall be of type real.

B shall have the same type and type parameters as A.

4 **Restriction.** IEEE_QUIET_EQ (A) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the value false.

5 **Result Characteristics.** Default logical.

6 **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 and no exception will be signaled.

7 **Example.** IEEE_QUIET_EQ (1.0, IEEE_VALUE (IEEE_QUIET_NAN)) has the value false and no exception is signaled.

14.11.25 IEEE_QUIET_GE (A, B)

1 **Description.** Quiet compares greater than or equal.

2 **Class.** Elemental function.

3 **Arguments.**

A shall be of type real.

B shall have the same type and type parameters as A.

4 **Restriction.** IEEE_QUIET_GE (A) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the value false.

5 **Result Characteristics.** Default logical.

6 **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 and no exception will be signaled.

7 **Example.** IEEE_QUIET_GE (1.0, IEEE_VALUE (IEEE_QUIET_NAN)) has the value false and no exception is signaled.

14.11.26 IEEE_QUIET_GT (A, B)

1 **Description.** Quiet compares greater than.

2 **Class.** Elemental function.

3 **Arguments.**

A shall be of type real.

B shall have the same type and type parameters as A.

4 **Restriction.** IEEE_QUIET_GT (A) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the value false.

5 **Result Characteristics.** Default logical.

6 **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 and no exception will be signaled.

7 **Example.** IEEE_QUIET_GT (1.0, IEEE_VALUE (IEEE_QUIET_NAN)) has the value false and no exception is signaled.

14.11.27 IEEE_QUIET_LE (A, B)

1 **Description.** Quiet compares less than or equal.

2 **Class.** Elemental function.

3 **Arguments.**

A shall be of type real.

B shall have the same type and type parameters as A.

4 **Restriction.** IEEE_QUIET_LE (A) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the value false.

5 **Result Characteristics.** Default logical.

6 **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 and no exception will be signaled.

7 **Example.** IEEE_QUIET_LE (1.0, IEEE_VALUE (IEEE_QUIET_NAN)) has the value false and no exception is signaled.

14.11.28 IEEE_QUIET_LT (A, B)

1 **Description.** Quiet compares less than.

2 **Class.** Elemental function.

3 **Arguments.**

A shall be of type real.

B shall have the same type and type parameters as A.

4 **Restriction.** IEEE_QUIET_LT (A) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the value false.

5 **Result Characteristics.** Default logical.

6 **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 and no exception will be signaled.

7 **Example.** IEEE_QUIET_LT (1.0, IEEE_VALUE (IEEE_QUIET_NAN)) has the value false and no exception is signaled.

14.11.29 IEEE_QUIET_NE (A, B)

1 **Description.** Quiet compares not equal.

2 **Class.** Elemental function.

3 **Arguments.**

A shall be of type real.

B shall have the same type and type parameters as A.

4 **Restriction.** IEEE_QUIET_NE (A) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the value false.

5 **Result Characteristics.** Default logical.

6 **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 and no exception will be signaled.

7 **Example.** IEEE_QUIET_NE (1.0, IEEE_VALUE (IEEE_QUIET_NAN)) has the value true and no exception is signaled.

14.11.30 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.

14.11.31 IEEE_REM (X, Y)

1 **Description.** Exact remainder.

2 **Class.** Elemental function.

3 **Arguments.** The arguments shall be of type real.

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*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.

14.11.32 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.

6 **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.

7 **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.

14.11.33 IEEE_SCALB (X, I)

1 **Description.** $X \times 2^I$.

2 **Class.** Elemental function.

3 **Arguments.**

X	shall be of type real.
I	shall be of type integer.

4 **Restriction.** IEEE_SCALB (X) shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the value false.

5 **Result Characteristics.** Same as X.

6 **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.

7 **Example.** The value of IEEE_SCALB (1.0, 2) is 4.0.

14.11.34 IEEE_SELECTED_REAL_KIND ([P, R, RADIX])

1 **Description.** IEEE kind type parameter value.

2 **Class.** Transformational function.

3 **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.

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. 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.

6 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.

7 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.

8 **Example.** IEEE_SELECTED_REAL_KIND (6, 30) has the value KIND (0.0) on a machine that supports ISO/IEC/IEEE 60559:2011 single precision arithmetic for its default real approximation method.

14.11.35 IEEE_SET_FLAG (FLAG, FLAG_VALUE)

1 **Description.** Set an exception flag.

2 **Class.** Pure subroutine.

3 **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.

4 **Example.** CALL IEEE_SET_FLAG (IEEE_OVERFLOW, .TRUE.) sets the IEEE_OVERFLOW flag to be signaling.

14.11.36 IEEE_SET_HALTING_MODE (FLAG, HALTING)

1 **Description.** Set a halting mode.

2 **Class.** Pure subroutine.

3 **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 true, the corresponding exception specified by FLAG will cause halting. Otherwise, execution will continue after this exception.

4 **Restriction.** IEEE_SET_HALTING_MODE (FLAG, HALTING) shall not be invoked if IEEE_SUPPORT_HALTING (FLAG) has the value false.

5 **Example.** CALL IEEE_SET_HALTING_MODE (IEEE_DIVIDE_BY_ZERO, .TRUE.) causes halting after a divide_by_zero exception.

14.11.37 IEEE_SET_MODES (MODES)

1 **Description.** Set floating-point modes.

2 **Class.** Subroutine.

3 **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 (14.7) are restored to the state at that invocation.

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.
```

14.11.38 IEEE_SET_ROUNDING_MODE (ROUND_VALUE [, RADIX])

1 **Description.** Set rounding mode.

2 **Class.** Subroutine.

3 **Arguments.**

ROUND_VALUE shall be a scalar of type IEEE_ROUND_TYPE. It specifies the 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.

4 **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.

5 **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
```

14.11.39 IEEE_SET_STATUS (STATUS_VALUE)

1 **Description.** Restore floating-point state.

2 **Class.** Subroutine.

3 **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 (14.7 is restored to the state at that invocation).

4 **Example.** To store all the exceptions flags, do a calculation involving exception handling, and restore them later:

```
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
```

14.11.40 IEEE_SET_UNDERFLOW_MODE (GRADUAL)

1 **Description.** Set underflow mode.

2 **Class.** Subroutine.

3 **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.

4 **Restriction.** IEEE_SET_UNDERFLOW_MODE shall not be invoked unless IEEE_SUPPORT_UNDERFLOW_-CONTROL (X) is true for some X.

5 **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)
```

14.11.41 IEEE_SIGNBIT (X)

1 **Description.** Test sign bit.

2 **Class.** Elemental function.

3 **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.

14.11.42 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 14.9.

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.

14.11.43 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 $f \neq 0$, 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_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 14.10

A reference to IEEE_SUPPORT_DENORMAL will have the same result value as a reference to IEEE_SUPPORT_SUBNORMAL with the same argument list.

14.11.44 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 conforms to ISO/IEC/IEEE 60559:2011.

14.11.45 IEEE_SUPPORT_FLAG (FLAG) or IEEE_SUPPORT_FLAG (FLAG, X)

1 **Description.** Query exception support.

2 **Class.** Inquiry 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.

14.11.46 IEEE_SUPPORT_HALTING (FLAG)

1 **Description.** Query halting mode support.

2 **Class.** Inquiry 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).

6 **Example.** IEEE_SUPPORT_HALTING (IEEE_OVERFLOW) has the value true if the processor supports control of halting after an overflow.

14.11.47 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.

6 **Example.** IEEE_SUPPORT_INF (X) has the value true if the processor supports IEEE infinities for X.

14.11.48 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 (9.5.6.16, 9.6.2.13, 10.7.2.3.8) 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.

6 **Example.** IEEE_SUPPORT_IO (X) has the value true if formatted input/output base conversions conform to ISO/IEC/IEEE 60559:2011.

14.11.49 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.

6 **Example.** IEEE_SUPPORT_NAN (X) has the value true if the processor supports IEEE NaNs for X.

14.11.50 IEEE_SUPPORT_ROUNDING (ROUND_VALUE) or IEEE_SUPPORT_ROUNDING (ROUND_VALUE, X)

1 **Description.** Query IEEE rounding support.

2 **Class.** Inquiry 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.

6 **Example.** IEEE_SUPPORT_ROUNDING (IEEE_TO_ZERO) has the value true if the processor supports rounding to zero for all reals.

14.11.51 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.

6 **Example.** If IEEE_SUPPORT_SQRT (1.0) has the value true, SQRT (-0.0) will have the value -0.0.

14.11.52 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.

5 **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.

6 **Example.** IEEE_SUPPORT_STANDARD () has the value false if some but not all kinds of reals conform to ISO/IEC/IEEE 60559:2011.

14.11.53 IEEE_SUPPORT_SUBNORMAL () or IEEE_SUPPORT_SUBNORMAL (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_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.

6 **Example.** IEEE_SUPPORT_SUBNORMAL (X) has the value true if the processor supports subnormal values for X.

NOTE 14.11

The subnormal numbers are not included in the 13.4 model for real numbers; they satisfy the inequality ABS (X) < TINY (X). They usually occur as a result of an arithmetic operation whose exact result is less than 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.

14.11.54 IEEE_SUPPORT_UNDERFLOW_CONTROL () or IEEE_SUPPORT_UNDERFLOW_CONTROL (X)

1 **Description.** Query underflow control 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_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.

6 **Example.** IEEE_SUPPORT_UNDERFLOW_CONTROL (2.5) has the value true if the processor supports underflow mode control for default real calculations.

14.11.55 IEEE_UNORDERED (X, Y)

1 **Description.** Whether two values are unordered.

2 **Class.** Elemental function.

3 **Arguments.** The arguments shall be of type real.

4 **Restriction.** IEEE_UNORDERED (X, Y) shall not be invoked if IEEE_SUPPORT_DATATYPE (X) or IEEE_-SUPPORT_DATATYPE (Y) has the value false.

5 **Result Characteristics.** Default logical.

6 **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.

7 **Example.** IEEE_UNORDERED (0.0, SQRT (−1.0)) has the value true if IEEE_SUPPORT_SQRT (1.0) has the value true.

14.11.56 IEEE_VALUE (X, CLASS)

1 **Description.** Return number in a class.

2 **Class.** Elemental function.

3 **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.

4 **Restriction.** IEEE_VALUE (X, CLASS) shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the value false.

5 **Result Characteristics.** Same as X.

6 **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.

7 **Example.** IEEE_VALUE (1.0, IEEE_NEGATIVE_INF) has the value −infinity.

8 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 14.12

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.

14.12 Examples

NOTE 14.13

```
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.
```

NOTE 14.13 (cont.)

```
   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 14.14

```
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_STATUS (STATUS_VALUE)
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.)
! 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)
```

NOTE 14.14 (cont.)

```
         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 14.15

```
      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.

15 Interoperability with C

15.1 General

1 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.

2 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 (4.5) and enumerations (4.6) that correspond to C types.

3 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.

15.2 The ISO_C_BINDING intrinsic module

15.2.1 Summary of contents

1 The processor shall provide the intrinsic module ISO_C_BINDING. This module shall make accessible the following entities: the named constants C_NULL_PTR and C_NULL_FUNPTR and those with names listed in the first column of Table 15.1 and the second column of Table 15.2, and the types C_PTR and C_FUNPTR. A processor may provide other public entities in the ISO_C_BINDING intrinsic module in addition to those listed here.

15.2.2 Named constants and derived types in the module

1 The entities listed in the second column of Table 15.2 shall be default integer named constants.

2 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.

3 The value of C_INT shall be a valid value for an integer kind 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 defines the corresponding C type and there is no interoperating Fortran processor kind or -2 if the C processor does not define the corresponding C type.

4 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.

5 The value of C_BOOL shall be a valid value for a logical kind parameter on the processor or shall be −1.

6 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 non-negative, the character kind specified is the C character kind; otherwise, there is no C character kind.

7 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 15.1. 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 15.1: **Names of C characters with special semantics**

Name	C definition	Value	
		C_CHAR = −1	C_CHAR ≠ −1
C_NULL_CHAR	null character	CHAR(0)	`'\0'`
C_ALERT	alert	ACHAR(7)	`'\a'`
C_BACKSPACE	backspace	ACHAR(8)	`'\b'`
C_FORM_FEED	form feed	ACHAR(12)	`'\f'`
C_NEW_LINE	new line	ACHAR(10)	`'\n'`
C_CARRIAGE_RETURN	carriage return	ACHAR(13)	`'\r'`
C_HORIZONTAL_TAB	horizontal tab	ACHAR(9)	`'\t'`
C_VERTICAL_TAB	vertical tab	ACHAR(11)	`'\v'`

8 The entities C_PTR and C_FUNPTR are described in 15.3.3.

9 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 15.1

The value of NEW_LINE(C_NEW_LINE) is C_NEW_LINE (13.7.122).

15.2.3 Procedures in the module

15.2.3.1 General

1 In the detailed descriptions below, procedure names are generic and not specific.

15.2.3.2 C_ASSOCIATED (C_PTR_1 [, C_PTR_2])

1 **Description.** Query C pointer status.

2 **Class.** Inquiry function.

3 **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 15.2

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)
```

15.2.3.3 C_F_POINTER (CPTR, FPTR [, SHAPE])

1 **Description.** Associate a data pointer with the target of a C pointer and specify its shape.

2 **Class.** Subroutine.

3 **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.

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. In this case, FPTR becomes pointer associated with the target of CPTR. If FPTR is an array, its shape is specified by SHAPE and each lower bound is 1.

Case (ii): If the value of CPTR is the result of a reference to C_LOC with a noninteroperable argument X, FPTR shall be a nonpolymorphic scalar pointer with the same type and type parameters as X. In this case, X or its target if it is a pointer shall not have been deallocated or have become undefined due to execution of a RETURN or END statement since the reference. FPTR becomes pointer associated with X or its target.

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 15.3

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.

15.2.3.4 C_F_PROCPOINTER (CPTR, FPTR)

1 **Description.** Associate a procedure pointer with the target of a C function pointer.

2 **Class.** 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 15.4

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.

15.2.3.5 C_FUNLOC (X)

1 **Description.** C address of the argument.

2 **Class.** Inquiry 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 FPTR. 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 FPTR%PX => X were executed.

6 The result is a value that can be used as an actual FPTR argument in a call to C_F_PROCPOINTER where FPTR 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.

15.2.3.6 C_LOC (X)

1 **Description.** C address of the argument.

2 **Class.** Inquiry function.

3 **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.

4 **Result Characteristics.** Scalar of type C_PTR.

5 **Result Value.** The result value is described using the result name CPTR.

6 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.

7 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.

8 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.

9 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 15.5

> 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.

15.2.3.7 C_SIZEOF (X)

1 **Description.** Size of X in bytes.

2 **Class.** Inquiry function.

3 **Argument.** X shall be an interoperable data entity that is not an assumed-size array.

4 **Result Characteristics.** Scalar integer of kind C_SIZE_T (15.3.2).

5 **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.

6 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.

15.3 Interoperability between Fortran and C entities

15.3.1 General

1 Subclause 15.3 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 15.6

> A Fortran entity can be interoperable with more than one C entity.

15.3.2 Interoperability of intrinsic types

1 Table 15.2 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.

2 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.

3 A combination of intrinsic type and type parameters is interoperable if it is interoperable with a C type. The C types mentioned in table 15.2 are defined in subclauses 6.2.5, 7.19, and 7.20.1 of ISO/IEC 9899:2011.

Table 15.2: **Interoperability between Fortran and C types**

Fortran type	Named constant from the ISO_C_BINDING module (kind type parameter if value is positive)	C type
INTEGER	C_INT	int
	C_SHORT	short int
	C_LONG	long int
	C_LONG_LONG	long long int
	C_SIGNED_CHAR	signed char unsigned char
	C_SIZE_T	size_t
	C_INT8_T	int8_t
	C_INT16_T	int16_t
	C_INT32_T	int32_t
	C_INT64_T	int64_t
	C_INT_LEAST8_T	int_least8_t
	C_INT_LEAST16_T	int_least16_t
	C_INT_LEAST32_T	int_least32_t
	C_INT_LEAST64_T	int_least64_t
	C_INT_FAST8_T	int_fast8_t
	C_INT_FAST16_T	int_fast16_t
	C_INT_FAST32_T	int_fast32_t
	C_INT_FAST64_T	int_fast64_t
	C_INTMAX_T	intmax_t
	C_INTPTR_T	intptr_t
	C_PTRDIFF_T	ptrdiff_t
REAL	C_FLOAT	float
	C_DOUBLE	double
	C_LONG_DOUBLE	long double
COMPLEX	C_FLOAT_COMPLEX	float _Complex
	C_DOUBLE_COMPLEX	double _Complex
	C_LONG_DOUBLE_COMPLEX	long double _Complex
LOGICAL	C_BOOL	_Bool
CHARACTER	C_CHAR	char

NOTE 15.7

For example, the type integer with a kind type parameter of C_SHORT is interoperable with the C type

NOTE 15.7 (cont.)

> short or any C type derived (via typedef) from short.

NOTE 15.8

> 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.

15.3.3 Interoperability with C pointer types

1 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 15.9

> 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 15.10

> 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.

15.3.4 Interoperability of derived types and C struct types

1 Interoperability between a derived type in Fortran and a struct type in C is provided by the BIND attribute on the Fortran type.

C1501 (R426) A derived type with the BIND attribute shall not have the SEQUENCE attribute.

C1502 (R426) A derived type with the BIND attribute shall not have type parameters.

C1503 (R426) A derived type with the BIND attribute shall not have the EXTENDS attribute.

C1504 (R426) A *derived-type-def* that defines a derived type with the BIND attribute shall not have a *type-bound-procedure-part*.

C1505 (R426) A derived type with the BIND attribute shall have at least one component.

C1506 (R426) Each component of a derived type with the BIND attribute shall be a nonpointer, nonallocatable data component with interoperable type and type parameters.

NOTE 15.11

> 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.

2 A derived type is interoperable with a C struct type if and only if the derived type has the BIND attribute (4.5.2), the derived type and the C struct type have the same number of components, and the components of the derived type would interoperate with corresponding components of the C struct type as described in 15.3.5 and 15.3.6 if the components were variables. A component of a derived type and a component of a C struct type correspond if they are declared in the same relative position in their respective type definitions.

NOTE 15.12

> The names of the corresponding components of the derived type and the C struct type need not be the same.

3 There is no Fortran type that is interoperable with a C struct type that contains a bit field or that contains a flexible array member. There is no Fortran type that is interoperable with a C union type.

NOTE 15.13

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 15.14

> 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 a Fortran derived type is interoperable with a C struct type because the case of identifiers is significant in C but not in Fortran.

15.3.5 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.

15.3.6 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 15.15

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 15.16

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 15.17

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 15.18

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 (12.5.2.11) 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 15.22 has an example of interoperation between Fortran and C strings.

15.3.7 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_desc_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_desc_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 15.19

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 15.20

The C language allows specification of a C function that can take a variable number of arguments (ISO/IEC 9899:2011, 7.16). This part of ISO/IEC 1539 does not provide a mechanism for Fortran procedures to interoperate with such C functions.

3 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.

4 In a reference from C to a procedure with an interoperable interface, if a dummy argument is allocatable, assumed-shape, assumed-rank, or a pointer, the corresponding actual argument in C shall be the address of a C descriptor for the actual argument. In this C descriptor, the members other than `attribute` and `type` shall describe an object with the same characteristics as the actual 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 from Table 15.4 that depends on the effective argument as follows:

- if the dynamic type of the effective argument is an interoperable type listed in Table 15.4, the corresponding value for that type;
- if the dynamic type of the effective argument is an intrinsic type with no corresponding type listed in Table 15.4, or a noninteroperable derived type that does not have type parameters, type-bound procedures, final subroutines, nor components that have the ALLOCATABLE or POINTER attributes, or correspond to CFI_type_other, one of the processor-dependent nonnegative type specifier values;
- otherwise, CFI_type_other.

5 In an invocation of an interoperable procedure whose Fortran interface has an assumed-shape or assumed-rank dummy argument with the CONTIGUOUS attribute, the associated effective argument may be an array that is not contiguous or the address of a C descriptor for such an array. If the procedure is invoked from Fortran or the procedure is a Fortran procedure, the Fortran processor will handle the difference in contiguity. If the procedure is invoked from C and the procedure is a C procedure, the C code within the procedure shall be prepared to handle the situation of receiving a discontiguous argument.

Unresolved Technical Issue 005

An effective argument is not EVER an address.

The concept of effective argument is a Fortran concept.

The concept of a C descriptor is not anything that exists in Fortran, but a *descriptor* of something that might exist in Fortran.

This probably needs to be split into two paragraphs, one for C calling Fortran and the other for Fortran calling C.

6 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 15.21

For example, a Fortran procedure interface described by

```
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

```
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 15.22

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.

NOTE 15.22 (cont.)

For example, consider a C function that can be described by the C function prototype

```
void copy(char in[], char out[]);
```

Such a function can be invoked from Fortran as follows:

```
  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 (12.5.2.11). 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.

15.4 C descriptors

1 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`.

15.5 The source file ISO_Fortran_binding.h

15.5.1 Summary of contents

1 The source file `ISO_Fortran_binding.h` shall contain the C structure definitions, typedef declarations, macro definitions, and function prototypes specified in subclauses 15.5.2 to 15.5.5. 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 an dummy argument that is allocatable, assumed-shape, assumed-rank, or a pointer.

2 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.

3 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.

15.5.2 The CFI_dim_t structure type

1 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 15.5.4. 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.

15.5.3 The CFI_cdesc_t structure type

1 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 part of ISO/IEC 1539, or received by a C function when invoked by a Fortran procedure, shall have the properties described in this subclause.

2 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 15.5.4. The type CFI_dim_t is described in 15.5.2.

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 15.4 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 15.3 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.

3 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.

4 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.

5 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 15.23

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 15.24

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.

15.5.4 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 15.25

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 (15.5.5.4).

```
CFI_CDESC_T(5) object;
int ind;
... code to define and use C descriptor ...
ind = CFI_deallocate((CFI_cdesc_t *)&object);
```

3 CFI_index_t is a typedef name for a standard signed integer type capable of representing the result of subtracting two pointers.

4 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.

5 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.

6 The macros in Table 15.3 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 15.3: **ISO_Fortran_binding.h macros for attribute codes**

Macro name	Attribute
CFI_attribute_pointer	data pointer
CFI_attribute_allocatable	allocatable
CFI_attribute_other	nonallocatable nonpointer

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 15.4 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 Additional nonnegative processor-dependent type specifier values may be defined for Fortran intrinsic types that are not represented by other type specifiers and noninteroperable Fortran derived types that do not have type parameters, type-bound procedures, final subroutines, allocatable components, or pointer components.

Unresolved Technical Issue 008

Is the processor *required* to define a nonnegative type specifier value for the above cases?

The words say “may”, i.e. is permitted to.

But the wording in 15.3.7 which apparently (see other UTI) is attempting to lay down requirements on the value of the `type` member of a CFI_desc_t, says

1. it “**shall** have a value from Table 15.4”; this seems unfortunate in itself, since that table does not even contain any such processor-dependent values.

2. “if the ... effective argument is [of] noninteroperable derived type ... [it shall have] one of the processor-dependent nonnegative type specifier values”; this would seem to require such values to exist, i.e. the operative word abvet should be “shall”.

This technical question needs answering before we can know *how* to fix the wording.

10 CFI_type_t is a typedef name for a standard integer type capable of representing the values for the supported type specifiers.

Table 15.4: **ISO_Fortran_binding.h macros for type codes**

Macro name	C Type
CFI_type_signed_char	signed char
CFI_type_short	short int
CFI_type_int	int
CFI_type_long	long int

ISO_Fortran_binding.h macros for type codes (cont.)

Macro name	C Type
CFI_type_long_long	long long int
CFI_type_size_t	size_t
CFI_type_int8_t	int8_t
CFI_type_int16_t	int16_t
CFI_type_int32_t	int32_t
CFI_type_int64_t	int64_t
CFI_type_int_least8_t	int_least8_t
CFI_type_int_least16_t	int_least16_t
CFI_type_int_least32_t	int_least32_t
CFI_type_int_least64_t	int_least64_t
CFI_type_int_fast8_t	int_fast8_t
CFI_type_int_fast16_t	int_fast16_t
CFI_type_int_fast32_t	int_fast32_t
CFI_type_int_fast64_t	int_fast64_t
CFI_type_intmax_t	intmax_t
CFI_type_intptr_t	intptr_t
CFI_type_ptrdiff_t	ptrdiff_t
CFI_type_float	float
CFI_type_double	double
CFI_type_long_double	long double
CFI_type_float_Complex	float _Complex
CFI_type_double_Complex	double _Complex
CFI_type_long_double_Complex	long double _Complex
CFI_type_Bool	_Bool
CFI_type_char	char
CFI_type_cptr	void *
CFI_type_struct	interoperable C structure
CFI_type_other	Not otherwise specified

NOTE 15.26

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 15.5 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 15.5 indicate the error condition described.

Table 15.5: **ISO_Fortran_binding.h macros for error codes**

Macro name	Error condition
CFI_SUCCESS	No error detected.
CFI_ERROR_BASE_ADDR_NULL	The base address member of a C descriptor is a null pointer in a context that requires a non-null pointer value.
CFI_ERROR_BASE_ADDR_NOT_NULL	The base address member of a C descriptor is not a null pointer in a context that requires a null pointer value.
CFI_INVALID_ELEM_LEN	The value supplied for the element length member of a C descriptor is not valid.
CFI_INVALID_RANK	The value supplied for the rank member of a C descriptor is not

ISO_Fortran_binding.h macros for error codes (cont.)

Macro name	Error condition
	valid.
CFI_INVALID_TYPE	The value supplied for the type member of a C descriptor is not valid.
CFI_INVALID_ATTRIBUTE	The value supplied for the attribute member of a C descriptor is not valid.
CFI_INVALID_EXTENT	The value supplied for the extent member of a CFI_dim_t structure is not valid.
CFI_INVALID_DESCRIPTOR	A C descriptor is invalid in some way.
CFI_ERROR_MEM_ALLOCATION	Memory allocation failed.
CFI_ERROR_OUT_OF_BOUNDS	A reference is out of bounds.

15.5.5 Functions declared in ISO_Fortran_binding.h

15.5.5.1 Arguments and results of the functions

1 Some of the functions described in 15.5.5 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 15.5 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 15.5.5 could detect more than one error condition and an error condition is detected, which error condition is detected is processor dependent.

2 In function arguments representing subscripts, bounds, extents, or strides, the ordering of the elements is the same as the ordering of the elements of the `dim` member of a C descriptor.

3 Prototypes for these functions, or equivalent macros, are provided in the `ISO_Fortran_binding.h` file as described in 15.5.5. 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 15.27

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.

15.5.5.2 The CFI_address function

1 **Synopsis.** C address of an object described by a C descriptor.

```
void *CFI_address(const CFI_cdesc_t *dv, const CFI_index_t subscripts[]);
```

2 **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.

3 **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 `subscripts` argument would specify if used as subscripts. If the object is scalar, the result is its C address.

4 **Example.** If `dv` is the address of a C descriptor for the Fortran array `A` declared as

```
REAL(C_FLOAT) :: A(100, 100)
```

the following code calculates the C address of A(5, 10):

```
CFI_index_t subscripts[2];
float *address;
subscripts[0] = 4;
subscripts[1] = 9;
address = (float *) CFI_address(dv, subscripts );
```

15.5.5.3 The CFI_allocate function

1 **Synopsis.** Allocate memory for an object described by a C descriptor.

```
int CFI_allocate(CFI_cdesc_t *dv, const CFI_index_t lower_bounds[],
                 const CFI_index_t upper_bounds[], size_t elem_len);
```

2 **Formal Parameters.**

`dv` shall be the address of a C descriptor specifying the rank and type of the object. The `base_addr` member of the C descriptor shall be a null pointer. If the type is not a character type, the `elem_len` member shall specify the element length. The `attribute` member shall have a value of CFI_attribute_allocatable or CFI_attribute_pointer.

`lower_bounds` shall be the address of an array with at least `dv->rank` elements.

`upper_bounds` shall be the address of an array with at least `dv->rank` elements.

`elem_len` If the type specified in the C descriptor type is a Fortran character type, the value of `elem_len` shall be the storage size in bytes of an element of the object; otherwise, `elem_len` is ignored.

3 **Description.** Successful execution of CFI_allocate allocates memory for the object described by the C descriptor with the address `dv` using the same mechanism as the Fortran ALLOCATE statement, and assigns the address of that memory to `dv->base_addr`. The first `dv->rank` elements of the `lower_bounds` and `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 C descriptor. If the rank is zero, the `lower_bounds` and `upper_bounds` arguments are ignored. If the type specified in the 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 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

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

and the array is not allocated, the following code allocates it to be of shape [100, 500]:

```
CFI_index_t lower[2], upper[2];
int ind;
lower[0] = 1; lower[1] = 1;
upper[0] = 100; upper[1] = 500;
ind = CFI_allocate(dv, lower, upper, 0);
```

15.5.5.4 The CFI_deallocate function

1 **Synopsis.** Deallocate memory for an object described by a C descriptor.

```
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 (6.7.3).

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

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

and the array is allocated, the following code deallocates it:

```
        int ind;
        ind = CFI_deallocate(dv);
```

15.5.5.5 The CFI_establish function

1 **Synopsis.** Establish a C descriptor.

```
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 15.3. If it is CFI_attribute_allocatable, `base_addr` shall be a null pointer.
- `type` shall be one of the type codes in Table 15.4.
- `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 $0 \leq$ `rank` $\leq$ 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.

3 **Description.** Successful execution of CFI_establish updates the object with the address `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 `base_addr` is not a null pointer, it is for a nonallocatable entity that is a scalar or a contiguous array; if the `attribute` argument has the value CFI_attribute_pointer, the lower bounds of the object described by `dv` are set to zero. If `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 `base_addr` is the C address of a Fortran data object, the `type` and `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 `dv` is not modified.

4 **Result Value.** The result is an error indicator.

NOTE 15.28

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 15.29

A C descriptor with `attribute` CFI_attribute_other and `base_addr` a null pointer can be used as the argument `result` in calls to CFI_section or CFI_select_part, which will produce a C descriptor for a nonallocatable nonpointer data object.

5 **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.

```
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

```
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 `a_fortran` for an array of size 100 that can be used as the actual argument in an invocation of the subprogram from 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);
```

15.5.5.6 The CFI_is_contiguous function

1 **Synopsis.** Test contiguity of an array.

```
int CFI_is_contiguous(const CFI_cdesc_t * dv);
```

2 **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.

3 **Result Value.** The value of the result is 1 if the array described by `dv` is contiguous, and 0 otherwise.

NOTE 15.30

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.

15.5.5.7 The CFI_section function

1 **Synopsis.** Update a C descriptor for an array section for which each element is an element of a given array.

```
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[]);
```

2 **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.

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 `stride`, as follows.

The array section is equivalent to the Fortran array section SOURCE($sectsub_1$, $sectsub_2$, ... $sectsub_n$), 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 $stride_i$ is 1 if `stride` is a null pointer and `stride[`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.

`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.

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):

```
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).

```
CFI_index_t lower[2], upper[2], strides[2];
CFI_CDESC_T(1) section;
int ind;
lower[0] = source->dim[0].lower;
upper[0] = source->dim[0].lower + source->dim[0].extent - 1;
strides[0] = 1;
lower[1] = upper[1] = source->dim[1].lower + 41;
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);
```

15.5.5.8 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.

```
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 $0 \leq$ `displacement` $\leq$ `source->elem_len` -1, and the sum of the displacement and the size in bytes 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.

`elem_len` shall have a value equal to the storage size in bytes of an element of the array section if `result->type` specifies a Fortran character type; otherwise, the value of this parameter is ignored.

3 **Description.** Successful execution of CFI_select_part updates the `base_addr`, `dim`, and `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

```
TYPE, BIND(C) :: T
  REAL(C_DOUBLE) :: X
  COMPLEX(C_DOUBLE_COMPLEX) :: Y
END TYPE
TYPE(T) A(100)
```

the following code fragment establishes a C descriptor for the array `A%Y`:

```
typedef struct {
  double x; double _Complex y;
} t;
CFI_CDESC_T(1) component;
CFI_cdesc_t * comp_cdesc = (CFI_cdesc_t *)&component;
CFI_index_t extent[] = { 100 };
(void)CFI_establish(comp_cdesc, NULL, CFI_attribute_other, CFI_type_double_Complex,
                    sizeof(double _Complex), 1, extent);
(void)CFI_select_part(comp_cdesc, source, offsetof(t,y), 0);
```

15.5.5.9 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.

```
int CFI_setpointer(CFI_cdesc_t *result, CFI_cdesc_t *source,
                   const CFI_index_t lower_bounds[]);
```

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 corresponding values of the `elem_len`, `rank`, and `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.

3 **Result Value.** The result is an error indicator.

4 **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.

5 **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.

```
CFI_index_t lower_bounds[1];
int ind;
lower_bounds[0] = 0;
ind = CFI_setpointer(ptr, ptr, lower_bounds);
```

15.6 Restrictions on C descriptors

1 A C descriptor shall not be initialized, updated, or copied other than by calling the functions specified in 15.5.5.

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 15.31

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.

15.7 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.

4 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.

15.8 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 them 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 15.32

The following example illustrates how a C descriptor becomes undefined upon returning from a call to a C function.

```
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.

15.9 Interoperation with C global variables

15.9.1 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.

2 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.

3 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.

4 There does not have to be an associated C entity for a Fortran entity with the BIND attribute.

NOTE 15.33

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.

```
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;
```

15.9.2 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 (16.5.1.5) with that variable.

15.10 Interoperation with C functions

15.10.1 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.

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,
- 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 ISO/IEC 9899:2011.

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 invoked as a signal handler (ISO/IEC 9899:2011, 7.14.1).

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 (9.5.4).

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 (*)()`.

15.10.2 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.

C1507 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 15.34

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)
  ...
END SUBROUTINE C_SUB
```

NOTE 15.34 (cont.)

```
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.

15.10.3 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 (14.7) 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.

15.10.4 Asynchronous communication

1 Asynchronous communication for a Fortran variable 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 dependent.

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 15.35

Asynchronous communication can be used for nonblocking MPI calls such as MPI_IRECV and MPI_ISEND. For example,

```
    REAL :: BUF(100,100)
    ... ! Code that involves BUF
    BLOCK
      ASYNCHRONOUS :: BUF
      CALL MPI_IRECV(BUF,...REQ,...)
         ... ! Code that does not involve BUF
      CALL MPI_WAIT(REQ,...)
    END BLOCK
```

NOTE 15.35 (cont.)

```
    ... ! 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.

16 Scope, association, and definition

16.1 Scopes, identifiers, and entities

1 An entity is identified by an identifier.

2 The scope of

- a global identifier is a program (2.2.2),
- a local identifier is an inclusive scope,
- an identifier of a construct entity is that construct (7.2.4, 8.1), and
- an identifier of a statement entity is that statement or part of that statement (3.3),

excluding any nested scope where the identifier is treated as the identifier of a different entity (16.3, 16.4).

3 An entity may be identified by

- an image index (1.3),
- a name (1.3),
- a statement label (1.3),
- an external input/output unit number (9.5),
- an identifier of a pending data transfer operation (9.6.2.9, 9.7),
- a submodule identifier (11.2.3),
- a generic identifier (1.3), or
- a binding label (1.3).

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.

16.2 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 part of ISO/IEC 1539 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 16.1

An intrinsic module is not a program unit, so a global identifier can be the same as the name of an intrinsic module.

NOTE 16.2

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.

16.3 Local identifiers

16.3.1 Classes of local identifiers

1 Identifiers of entities in the classes

(1) except for statement or construct entities (16.4), named variables, named constants, 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 (16.3.2),
- 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 (16.3.3).

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 12.4.3.5 or the same as the name of a derived type (4.5.10). A local identifier of one class may be the same as a local identifier of another class.

NOTE 16.3

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)
   ...
CONTAINS
   FUNCTION SIN (X)
      ...
   END FUNCTION SIN
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.

4 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).

16.3.2 Local identifiers that are the same as common block names

1 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.

16.3.3 Function results

1 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.

16.3.4 Components, type parameters, and bindings

1 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.

2 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*.

3 The binding name (4.5.5) 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.

4 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.

5 A component name or binding name may appear only in a scope in which it is accessible.

6 The accessibility of components and bindings is specified in 4.5.4.8 and 4.5.5.

16.3.5 Argument keywords

1 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 (16.5.1.3, 16.5.1.4), the argument keyword is accessible for procedure references for that procedure in that scoping unit.

2 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.

16.4 Statement and construct entities

1 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 or as an *associate-name* in a SELECT TYPE or ASSOCIATE 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.

2 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.

3 The name of a statement entity shall not be the same as an accessible global identifier or local identifier of class (1) (16.3.1), 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.

4 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.

5 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.

6 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.

7 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 8.1.9.2.

8 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 8.1.3.2.

9 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.

16.5 Association

16.5.1 Name association

16.5.1.1 Forms of name association

1 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.

16.5.1.2 Argument association

1 The rules governing argument association are given in Clause 12. As explained in 12.5, execution of a procedure reference establishes a correspondence between each dummy argument and an actual argument and thus an association between each dummy argument and its effective argument. Argument association may be sequence association (12.5.2.11).

2 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 16.4

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.

3 Upon termination of execution of a procedure reference, all argument associations established by that reference are terminated. A dummy argument of that procedure may be associated with an entirely different effective argument in a subsequent invocation of the procedure.

16.5.1.3 Use association

1 Use association is the association of names in different scopes specified by a USE statement. The rules governing use association are given in 11.2.2. They allow for renaming of entities being accessed. Use association allows access in one scope to entities defined in another scope; it remains in effect throughout the execution of the program.

16.5.1.4 Host association

1 A nested scoping unit has access to named entities from its host as specified in 12.4.3.4. 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.

2 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 (5.5.9) 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 16.5

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.

3 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.

4 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 16.6

An interface body that is not a module procedure interface body accesses by host association only those entities made accessible by IMPORT statements.

5 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.

6 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 16.7

A host subprogram and an internal subprogram can contain the same and differing use-associated entities, as illustrated in the following example.

```
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
                      ! 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
```

NOTE 16.7 (cont.)

```
      ...
   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 16.8

For more examples of host association, see subclause C.11.1.

16.5.1.5 Linkage association

1 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 (15.9). Such association remains in effect throughout the execution of the program.

16.5.1.6 Construct association

1 Execution of a SELECT TYPE statement establishes an association between the selector and the associate name of the construct. Execution of an ASSOCIATE statement establishes an association between each selector and the corresponding associate name of the construct.

2 If the selector is allocatable, it shall be allocated; the associate name is associated with the data object and does not have the ALLOCATABLE attribute.

3 If the 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.

4 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.

5 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 16.9

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*.

16.5.2 Pointer association

16.5.2.1 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.

16.5.2.2 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 16.10

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 part of ISO/IEC 1539 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.

16.5.2.3 Events that cause pointers to become associated

1 A pointer becomes associated when any of the following events occur.

(1) The pointer is allocated (6.7.1) as the result of the successful execution of an ALLOCATE statement referencing the pointer.

(2) The pointer is pointer-assigned to a target (7.2.2) 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.

16.5.2.4 Events that cause pointers to become disassociated

1 A pointer becomes disassociated when

(1) the pointer is nullified (6.7.2),

(2) the pointer is deallocated (6.7.3),

(3) the pointer is pointer-assigned (7.2.2) 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.

16.5.2.5 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 (13.7.119) is executed, the pointer is associated with the argument FROM, and an object without the TARGET attribute is pointer associated with the argument TO,

(6) completion of execution of an instance of a subprogram causes the pointer's target to become undefined (item (3) of 16.6.6),

(7) completion of execution of a BLOCK construct causes the pointer's target to become undefined (item (22) of 16.6.6),

(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 unless the pointer

  (a) has the SAVE attribute,

  (b) is in blank common,

  (c) is in a named common block that is declared in at least one other scoping unit that is in execution,

  (d) is accessed by host association, or

  (e) is the return value 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) 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,

(14) 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), or

(15) a procedure is invoked with the pointer as an actual argument corresponding to a pointer dummy argument with INTENT (OUT).

16.5.2.6 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 16.5.5.

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.

16.5.2.7 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, the definition status of the pointer may be defined or undefined according to the rules for a variable (16.6). The definition status of a pointer that is not associated is undefined.

16.5.2.8 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 deallocated. 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*.

16.5.3 Storage association

16.5.3.1 General

1 Storage sequences are used to describe relationships that exist among variables and common blocks. Storage association 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.

16.5.3.2 Storage sequence

1 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 (9.3.5), 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 (13.8.2).

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 character with the C character kind (15.2.2) and character length *len* occupies *len* contiguous unspecified storage units,

(5) a nonpointer scalar object of sequence type with no type parameters 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 (6.5.3.2), and

(8) 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.

3 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.

4 Each common block has a storage sequence (5.9.2.2).

16.5.3.3 Association of storage sequences

1 Two nonzero-sized storage sequences s_1 and s_2 are storage associated if the ith storage unit of s_1 is the same as the jth 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 $1 \leq i+k \leq \mathit{size\ of}\ s_1$ and $1 \leq j+k \leq \mathit{size\ of}\ s_2$ where *size of* measures the number of storage units.

2 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.

16.5.3.4 Association of scalar data objects

1 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.

2 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.

3 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 (5.9.1.2, 5.9.2.2).

4 COMMON statements cause data objects in one scoping unit to become storage associated with data objects in another scoping unit.

5 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.

6 An ENTRY statement in a function subprogram causes storage association of the function results that are variables.

7 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.

8 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.

9 Partial association of character entities occurs when some, but not all, of the storage units of the entities are the same.

10 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 (4.5.4.6). 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.

16.5.4 Inheritance association

1 Inheritance association occurs between components of the parent component and components inherited by type extension into an extended type (4.5.7.2). This association is persistent; it is not affected by the accessibility of the inherited components.

16.5.5 Establishing associations

1 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.

2 For argument association, the pre-existing entity is the effective argument and the associating entity is the dummy argument.

3 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 (12.6.2.4). Otherwise the pre-existing entity that participates in the association is the entity in the host scoping unit.

4 For construct association, the associating entity is identified by the associate name and the pre-existing entity is the selector.

5 When an association is established by argument association, host association, or construct association, the following 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 deferred 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.

16.6 Definition and undefinition of variables

16.6.1 Definition of objects and subobjects

1 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.

2 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.

3 An array is defined if and only if all of its elements are defined.

4 A derived-type scalar object is defined if and only if all of its nonpointer components are defined.

5 A complex or character scalar object is defined if and only if all of its subobjects are defined.

6 If an object is undefined, at least one (but not necessarily all) of its subobjects are undefined.

16.6.2 Variables that are always defined

1 Zero-sized arrays and zero-length strings are always defined.

16.6.3 Variables that are initially defined

1 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 ALLOCATABLE 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 16.11

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.

16.6.4 Variables that are initially undefined

1 All other variables are initially undefined.

16.6.5 Events that cause variables to become defined

1 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 (7.2.3).

(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 16.6.6.) 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 the entire dummy argument data object to become defined if the dummy argument 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 (9.7.1) 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 causes the array 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 object of zero size 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) An object with the VOLATILE attribute that is changed by a means not specified by the program might become defined (see 5.5.19).

(27) 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.

(28) Execution of an OPEN statement containing a NEWUNIT= specifier causes the specified integer variable to become defined.

(29) 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.

(30) Successful execution of a LOCK statement that does not contain an ACQUIRED_LOCK= specifier causes the lock variable to become defined.

(31) Successful execution of an UNLOCK statement causes the lock variable to become defined.

16.6.6 Events that cause variables to become undefined

1 Variables become undefined by the following events.

(1) When a scalar variable of intrinsic type becomess 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 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*s, all of the *do-variable*s in the statement become undefined (9.11).

(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 (9.10).

(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 (13.7.119) 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 (16.5.2.4, 16.5.2.5), 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) 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 (9.7.1).

(19) When a variable with the TARGET attribute is deallocated, a variable of type C_PTR becomes undefined if its value is the C address of any part of the variable that is deallocated.

(20) When a pointer is deallocated, a variable of type C_PTR becomes undefined if its value is the C address of any part of the target that is deallocated.

(21) Execution of the allocation transfer procedure (13.7.125) where an object without the TARGET attribute is pointer associated with the argument TO causes a variable of type C_PTR to become undefined if its value is the C address of any part of the argument FROM.

(22) When a BLOCK construct completes execution,

- its unsaved local variables become undefined, and
- a variable of type C_PTR whose value is the C address of an unsaved local variable of the BLOCK construct becomes undefined.

(23) When execution of the host instance of the target of a variable of type C_FUNPTR is completed by execution of a RETURN or END statement, the variable becomes undefined.

(24) Execution of an intrinsic assignment of the type C_PTR or C_FUNPTR in which the variable and *expr* are not on the same image causes the variable to become undefined.

(25) An object with the VOLATILE attribute (5.5.19) might become undefined by means not specified by the program.

(26) When a pointer becomes associated with a target that is undefined, the pointer becomes undefined.

NOTE 16.12

Execution of a defined assignment statement could leave all or part of the variable undefined.

16.6.7 Variable definition context

1 Some variables are prohibited from appearing in a syntactic context that would imply definition or undefinition of the variable (5.5.10, 5.5.15, 12.7). 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 a SELECT TYPE or ASSOCIATE construct if the corresponding associate name appears in a variable definition context;

(12) a *lock-variable* in a LOCK or UNLOCK statement;

(13) a *scalar-logical-variable* in an ACQUIRED_LOCK= specifier.

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.

16.6.8 Pointer association context

1 Some pointers are prohibited from appearing in a syntactic context that would imply alteration of the pointer association status (16.5.2.2,5.5.10, 5.5.15). 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

1 This part of ISO/IEC 1539 does not specify the following:

- the properties excluded in 1.1;
- a processor's error detection capabilities beyond those listed in 1.5;
- which additional intrinsic procedures or modules a processor provides (1.5);
- the number and kind of companion processors (2.5.7);
- the number of representation methods and associated kind type parameter values of the intrinsic types (4.4), 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

1 According to this part of ISO/IEC 1539, the following are processor dependent:

- the order of evaluation of the specification expressions within the specification part of an invoked Fortran procedure (2.3.5);
- how soon an image terminates if another image initiates error termination (2.3.5);
- whether the processor supports a concept of process exit status, and if so, the process exit status on program termination (2.3.6);
- the mechanism of a companion processor, and the means of selecting between multiple companion processors (2.5.7);
- the processor character set (3.1);
- the means for specifying the source form of a program unit (3.3);
- the maximum number of characters allowed on a source line containing characters not of default kind (3.3.2, 3.3.3);
- the maximum depth of nesting of include lines (3.4);
- the interpretation of the *char-literal-constant* in the include line (3.4);
- the set of values supported by an intrinsic type, other than logical (4.1.3);
- the kind of a character length type parameter (4.4.4.1);
- the blank padding character for nondefault character kind (4.4.4.2)
- whether particular control characters can appear within a character literal constant in fixed source form (4.4.4.3);
- the collating sequence for each character set (4.4.4.4);
- the order of finalization of components of objects of derived type (4.5.6.2);
- the order of finalization when several objects are finalized as the consequence of a single event (4.5.6.2);

- 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 (4.5.6.3);
- whether an object is finalized by a deallocation in which an error condition occurs (4.5.6.3);
- the kind type parameter of each enumeration and its enumerators (4.6);
- whether an array is contiguous, except as specified in 5.5.7;
- the set of error conditions that can occur in ALLOCATE and DEALLOCATE statements (6.7.1, 6.7.3);
- 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 (6.7.1.3);
- the order of deallocation when several objects are deallocated by a DEALLOCATE statement (6.7.3);
- the order of deallocation when several objects are deallocated due to the occurence of an event described in 6.7.3.2;
- whether an allocated allocatable subobject is deallocated when an error condition occurs in the deallocation of an object (6.7.3.2);
- the positive integer values assigned to the *stat-variable* in a STAT= specifier as the result of an error condition (6.7.4, 8.5.7);
- the allocation status or pointer association status of an *allocate-object* if an error occurs during execution of an ALLOCATE or DEALLOCATE statement (6.7.4);
- the value assigned to the *errmsg-variable* in an ERRMSG= specifier as the result of an error condition (6.7.5, 8.5.7);
- 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 (7.1.9.3);
- 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 (7.2.1.3);
- the order of evaluation of the specification expressions within the specification part of a BLOCK construct when the construct is executed (8.1.4);
- 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 (8.1.6);
- 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 (8.1.6);
- the manner in which the stop code of a STOP or ERROR STOP statement is made available (8.4);
- the mechanisms available for creating dependencies for cooperative synchronization (8.5.5);
- the set of error conditions that can occur in image control statements (8.5.7);
- 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 (9);
- whether particular control characters can appear in a formatted record or a formatted stream file (9.2.2);
- the form of values in an unformatted record (9.2.3);

- 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 (9.3);
- the set of allowable names for a file (9.3);
- whether a named file on one image is the same as a file with the same name on another image (9.3.1);
- the set of external files that exist for a program (9.3.2);
- the relationship between positions of successive file storage units in an external file that is connected for formatted stream access (9.3.3.4);
- 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 (9.5);
- 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 (9.5);
- the external unit preconnected for sequential formatted output and identified by the named constant ERROR_UNIT of the ISO_FORTRAN_ENV intrinsic module, and whether this unit is the same as OUTPUT_UNIT (9.5);
- at any time, the set of external units that exist for an image (9.5.3);
- whether a unit can be connected to a file that is also connected to a C stream (9.5.4);
- whether a file can be connected to more than one unit at the same time (9.5.4);
- the result of performing input/output operations on a unit connected to a file that is also connected to a C stream (9.5.4);
- 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 (9.5.4);
- the results of performing input/output operations on an external file both from Fortran and from a procedure defined by means other than Fortran (9.5.4);
- the default value for the ACTION= specifier on the OPEN statement (9.5.6.4);
- the encoding of a file opened with ENCODING='DEFAULT' (9.5.6.9);
- the file connected by an OPEN statement with STATUS='SCRATCH' (9.5.6.10);
- the interpretation of case in a file name (9.5.6.10, 9.10.2.2);
- 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 (9.5.6.14);
- the default value for the RECL= specifier in an OPEN statement (9.5.6.15);
- the effect of RECL= on a record containing any nondefault characters (9.5.6.15);
- the default input/output rounding mode (9.5.6.16);
- the default sign mode (9.5.6.17);
- the file status when STATUS='UNKNOWN' is specified in an OPEN statement (9.5.6.18);
- whether POS= is permitted with particular files, and whether POS= can position a particular file to a position prior to its current position (9.6.2.11);
- 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 (9.6.3);
- the result of unformatted input when the value stored in the file has a different type or type parameters from the input list item, as described in 9.6.4.5.2;
- the negative value of the `unit` argument to a defined input/output procedure if the parent data transfer statement accesses an internal file (9.6.4.8.3);

- 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 (9.6.4.8.3);
- 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 (9.9);
- the case of characters assigned to the variable in a NAME= specifier in an INQUIRE statement (9.10.2.15);
- the value of the variable in a POSITION= specifier in an INQUIRE statement if the file has been repositioned since connection (9.10.2.23);
- the relationship between file size and the data stored in records in a sequential or direct access file (9.10.2.30);
- the number of file storage units needed to store data in an unformatted file (9.10.3);
- the set of error conditions that can occur in input/output statements (9.11);
- when an input/output error condition occurs or is detected (9.11);
- the positive integer value assigned to the variable in an IOSTAT= specifier as the result of an error condition (9.11.5);
- the value assigned to the variable in an IOMSG= specifier as the result of an error condition (9.11.6);
- the result of output of non-representable characters to a Unicode file (10.7.1);
- the interpretation of the optional non-blank characters within the parentheses of a real NaN input field (10.7.2.3.2);
- the interpretation of a sign in a NaN input field (10.7.2.3.2);
- for output of an IEEE NaN, whether after the letters 'NaN', the processor produces additional alphanumeric characters enclosed in parentheses (10.7.2.3.2);
- the effect of the input/output rounding mode PROCESSOR_DEFINED (10.7.2.3.8);
- 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 (10.7.2.3.8);
- the field width, decimal part width, and exponent width used for the G0 edit descriptor (10.7.5);
- 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 (10.8.1);
- when the sign mode is PROCESSOR_DEFINED, whether a plus sign appears in a numeric output field for a nonnegative value (10.8.4);
- the results of list-directed output (10.10.4);
- the results of namelist output (10.11.4);
- the interaction between argument association and pointer association, (12.5.2.4);
- the values returned by some intrinsic functions (13);
- how the sequences of atomic actions in unordered segments interleave (13.1);
- the effects of calling COMMAND_ARGUMENT_COUNT, EXECUTE_COMMAND_LINE, GET_COMMAND, and GET_COMMAND_ARGUMENT on any image other than image 1 (13.5);
- whether each image uses a separate random number generator, or if some or all images use common random number generators (13.5);
- whether the results returned from CPU_TIME, DATE_AND_TIME and SYSTEM_CLOCK are dependent on which image calls them (13.5);
- the set of error conditions that can occur in some intrinsic subroutines (13.7);

- the value assigned to a CMDSTAT or STATUS argument to indicate a processor-dependent error condition (13.7);
- the value assigned to the TIME argument by the intrinsic subroutine CPU_TIME (13.7.43);
- whether date, clock, and time zone information is available (13.7.45);
- whether date, clock, and time zone information on one image is the same as that on another image (13.7.45);
- the value of command argument zero, if the processor does not support the concept of a command name (13.7.67);
- whether the significant length of a command argument includes trailing blanks (13.7.67);
- 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 (13.7.68);
- the computation of the seed value used by the pseudorandom number generator (13.7.139);
- on images that use a common random number generator, the interleaving of values assigned by RANDOM_NUMBER in unordered segments(13.5);
- the value assigned to the seed by the intrinsic subroutine RANDOM_SEED when no argument is present (13.7.139);
- the values assigned to its arguments by the intrinsic subroutine SYSTEM_CLOCK (13.7.167);
- the values of the named constants in the intrinsic module ISO_FORTRAN_ENV(13.8.2);
- the values returned by the functions COMPILER_OPTIONS and COMPILER_VERSION in the intrinsic module ISO_FORTRAN_ENV(13.8.2);
- the extent to which a processor supports IEEE arithmetic (14);
- the conditions under which IEEE_OVERFLOW is raised in a calculation involving non-ISO/IEC/IEEE 60559:2011 floating-point data;
- the conditions under which IEEE_OVERFLOW and IEEE_DIVIDE_BY_ZERO are raised in a floating-point exponentiation operation;
- the conditions under which IEEE_DIVIDE_BY_ZERO is raised in a calculation involving non-ISO/IEC/IEEE 60559:2011 floating-point data;
- the initial rounding modes (14.4);
- the initial underflow mode (14.5);
- the initial halting mode (14.6);
- the values of the floating-point exception flags on entry to a procedure defined by means other than Fortran (15.10.3),
- the value of CFI_MAX_RANK in the source file CFI_Fortran_binding.h (15.5.4);
- the value of CFI_VERSION in the source file CFI_Fortran_binding.h (15.5.4);
- which error condition is detected if more than one error condition is detected for an invocation of one of the functions declared in the source file CFI_Fortran_binding.h (15.5.5.1);
- the values of the attribute specifier macros defined in the source file CFI_Fortran_binding.h (15.5.4);
- 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 (15.5.4);
- the values of the error code macros other than CFI_SUCCESS that are defined in the source file CFI_Fortran_binding.h (15.5.4);
- the base address of a zero-sized array (15.5.3);
- the requirements on the storage sequence to be associated with the pointer FPTR by the C_F_POINTER

subroutine (15.2.3.4);

- whether a procedure defined by means other than Fortran is an asynchronous communication initiation or completion procedure (15.10.4).

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 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

1 These deleted features are those features of Fortran 2008 that were redundant and considered largely unused.

2 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

1 The obsolescent features are those features of Fortran 90 that were redundant and for which better methods were available in Fortran 90. Subclause 1.7.3 describes the nature of the obsolescent features. The obsolescent features in this part of ISO/IEC 1539 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.
(9) Label form of DO statement – see B.3.10.
(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

1 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,

```
CALL SUBR_NAME (X, Y, Z, *100, *200, *300)
```

can be replaced by

```
CALL SUBR_NAME (X, Y, Z, RETURN_CODE)
SELECT CASE (RETURN_CODE)
```

```
        CASE (1)
           ...
        CASE (2)
           ...
        CASE (3)
           ...
        CASE DEFAULT
           ...
     END SELECT
```

B.3.3 Computed GO TO statement

1 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

1 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.

2 The internal function is a more generalized form of the statement function and completely supersedes it.

B.3.5 DATA statements among executables

1 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

1 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.

2 Note that dummy arguments of a function can have assumed character length.

B.3.7 Fixed form source

1 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.

2 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

1 In addition to the CHARACTER**char-length* form introduced in FORTRAN 77, Fortran 90 provided the CHARACTER([LEN =] *type-param-value*) form. The older form (CHARACTER**char-length*) is redundant.

B.3.9 ENTRY statements

1 ENTRY statements allow more than one entry point to a subprogram, facilitating sharing of data items and executable statements local to that subprogram.

2 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

1 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

1 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

1 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

1 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 Clause 4 notes

C.1.1 Selection of the approximation methods (4.4.3.2)

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

```
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.

2 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 (13.7.152). 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.

3 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.1.2 Type extension and component accessibility (4.5.2.2, 4.5.4)

1 The default accessibility of an extended type can be specified in the type definition. The accessibility of its components can be specified individually. For example:

```
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
    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.1.3 Generic type-bound procedures (4.5.5)

Example of a derived type with generic type-bound procedures:

1 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.1.4 Abstract types (4.5.7.1)

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
```

2 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.1.5 Pointers (4.5.4.4, 5.5.14)

1 Pointers are names that can change dynamically their association with a target object. In a sense, a normal variable is a name with a fixed association with a particular object. A normal variable name refers to the same storage space throughout the lifetime of the variable. A pointer name can refer to different storage space, or even no storage space, at different times. A variable might be imagined to be a descriptor for space to hold values of the appropriate type, type parameters, and rank such that the values stored in the descriptor are fixed when the variable is created. A pointer also might be imagined to be a descriptor, but one whose values can be changed dynamically so as to describe different pieces of storage. When a pointer is declared, space to hold the descriptor is created, but the space for the target object is not created.

2 A derived type can have one or more components that are defined to be pointers. It can have a component that is a pointer to an object of the same derived type. This “recursive” data definition enables dynamic data structures such as linked lists, trees, and graphs to be constructed. For example:

```
TYPE NODE           ! Define a ''recursive'' type
   INTEGER :: VALUE = 0
   TYPE (NODE), POINTER :: NEXT_NODE => NULL ( )
END TYPE NODE

TYPE (NODE), TARGET :: HEAD         ! Automatically initialized
```

```
TYPE (NODE), POINTER :: CURRENT     ! Declare pointer
INTEGER :: IOEM, K

CURRENT => HEAD                     ! CURRENT points to head of list

DO
   READ (*, *, IOSTAT = IOEM) K     ! Read next value, if any
   IF (IOEM /= 0) EXIT
   ALLOCATE ( CURRENT % NEXT_NODE ) ! Create new cell
   CURRENT % NEXT_NODE % VALUE = K  ! Assign value to new cell
   CURRENT => CURRENT % NEXT_NODE   ! CURRENT points to new end of list
END DO
```

3 A list is now constructed and the last linked cell contains a disassociated pointer. A loop can be used to "walk through" the list.

```
CURRENT => HEAD
DO
   IF (.NOT. ASSOCIATED (CURRENT % NEXT_NODE)) EXIT
   CURRENT => CURRENT % NEXT_NODE
   WRITE (*, *) CURRENT % VALUE
END DO
```

C.1.6 Structure constructors and generic names (4.5.10)

1 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
  ! Operator definitions etc.
  ...
CONTAINS
  TYPE(mytype) FUNCTION int_to_mytype(i)
    INTEGER,INTENT(IN) :: i
    int_to_mytype%value = i
    int_to_mytype%exact = .TRUE.
  END FUNCTION
  ! Procedures to support operators etc.
  ...
END
```

```
PROGRAM example
  USE mytype_module
  TYPE(mytype) x
  x = mytype(17)
END
```

2 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
  INTEGER,PARAMETER :: single = KIND(0.0), double = KIND(0d0)
  INTERFACE t
    MODULE PROCEDURE real_to_t1, dble_to_t2, int_to_t1, int_to_t2
  END INTERFACE
  ...
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.1.7 Final subroutines (4.5.6, 4.5.6.2, 4.5.6.3, 4.5.6.4)

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)
    TYPE(t(KIND(0.0))) x
    IF (ASSOCIATED(x%vector)) DEALLOCATE(x%vector)
  END SUBROUTINE
  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
END MODULE

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
```

Example of extended types with final subroutines:

```
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
    TYPE(t2) :: x
    IF (ASSOCIATED(x%c)) DEALLOCATE(x%c)
    IF (ASSOCIATED(x%d)) DEALLOCATE(x%d)
  END SUBROUTINE
  SUBROUTINE t3f(y) ! Finalizer for TYPE(t3)'s extra components
    TYPE(t3) :: y
    IF (ASSOCIATED(y%e)) DEALLOCATE(y%e)
  END SUBROUTINE
END MODULE

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.2 Clause 5 notes

C.2.1 The POINTER attribute (5.5.14)

1 Specifying the POINTER attribute for an entity declares that it is a pointer. The type, type parameters, and rank, which can be specified in the same statement or with one or more attribute specification statements, determine the characteristics of the target objects that can be associated with the pointers declared in the statement. An obvious model for interpreting declarations of pointers is that such declarations create for each name a descriptor. Such a descriptor includes all the data necessary to describe fully and locate in memory an object and all subobjects of the type, type parameters, and rank specified. The descriptor is created empty; it does not contain values describing how to access an actual memory space. These descriptor values will be filled in when the pointer is associated with actual target space.

2 The following example illustrates the use of pointers in an iterative algorithm:

```
PROGRAM DYNAM_ITER
   REAL, DIMENSION (:, :), POINTER :: A, B, SWAP  ! Declare pointers
   ...
   READ (*, *) N, M
   ALLOCATE (A (N, M), B (N, M))  ! Allocate target arrays
   ! Read values into A
   ...
   ITER: DO
      ...
      ! Apply  transformation of values in A to produce values in B
      ...
      IF (CONVERGED) EXIT ITER
      ! Swap A and B
      SWAP => A; A => B; B => SWAP
   END DO ITER
   ...
END PROGRAM DYNAM_ITER
```

C.2.2 The TARGET attribute (5.5.17)

1 The TARGET attribute shall be specified for any nonpointer object that might, during the execution of the program, become associated with a pointer. This attribute is defined primarily for optimization purposes. It allows the processor to assume that any nonpointer object not explicitly declared as a target cannot be referenced by way of a pointer. It also means that implicitly-declared objects shall not be used as pointer targets. This will allow a processor to perform optimizations that otherwise would not be possible in the presence of certain pointers.

2 The following example illustrates the use of the TARGET attribute in an iterative algorithm:

```
PROGRAM ITER
   REAL, DIMENSION (1000, 1000), TARGET :: A, B
   REAL, DIMENSION (:, :), POINTER       :: IN, OUT, SWAP
   ...
   ! Read values into A
```

```
      ...
      IN => A            ! Associate IN with target A
      OUT => B           ! Associate OUT with target B
      ...
      ITER:DO
         ...
         ! Apply transformation of IN values to produce OUT
         ...
         IF (CONVERGED) EXIT ITER
         ! Swap IN and OUT
         SWAP => IN; IN => OUT; OUT => SWAP
      END DO ITER
      ...
   END PROGRAM ITER
```

C.2.3 The VOLATILE attribute (5.5.19)

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.3 Clause 6 notes

C.3.1 Structure components (6.4.2)

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

TYPE (LARGE) A (5)       ! 5 element array, each of whose elements
                         ! includes a 10 element array ELT and
                         ! a scalar VAL.
PRINT *, A (1)           ! Prints 10 element array ELT and scalar VAL.
PRINT *, A (1) % ELT (3) ! Prints scalar element 3
                         ! of array element 1 of A.
PRINT *, A (2:4) % VAL   ! Prints scalar VAL for array elements
                         ! 2 to 4 of A.
```

3 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:

```
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.3.2 Allocation with dynamic type (6.7.1)

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.

```
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.3.3 Pointer allocation and association (6.7.1, 16.5.2)

1 The effect of ALLOCATE, DEALLOCATE, NULLIFY, and pointer assignment is that they are interpreted as changing the values in the descriptor that is the pointer. An ALLOCATE is assumed to create space for a suitable object and to "assign" to the pointer the values necessary to describe that space. A NULLIFY breaks the association of the pointer with the space. A DEALLOCATE breaks the association and releases the space. Depending on the implementation, it could be seen as setting a flag in the pointer that indicates whether the values in the descriptor are valid, or it could clear the descriptor values to some (say zero) value indicative of the pointer not being associated with anything. A pointer assignment copies the values necessary to describe the space occupied by the target into the descriptor that is the pointer. Descriptors are copied; values of objects are not.

2 If PA and PB are both pointers and PB is associated with a target, then

```
PA => PB
```

results in PA being associated with the same target as PB. If PB was disassociated, then PA becomes disassociated.

3 This part of ISO/IEC 1539 is specified so that such associations are direct and independent. A subsequent statement

```
PB => D
```

or

```
ALLOCATE (PB)
```

has no effect on the association of PA with its target. A statement

```
DEALLOCATE (PB)
```

deallocates the space that is associated with both PA and PB. PB becomes disassociated, but there is no requirement that the processor make it explicitly recognizable that PA no longer has a target. This leaves PA as a "dangling pointer" to space that has been released. The program shall not use PA again until it becomes associated via pointer assignment or an ALLOCATE statement.

4 DEALLOCATE can only be used to release space that was created by a previous ALLOCATE. Thus the following is invalid:

```
REAL, TARGET :: T
REAL, POINTER :: P
   ...
P = >  T
DEALLOCATE (P) ! Not allowed: P's target was not allocated
```

5 The basic principle is that ALLOCATE, NULLIFY, and pointer assignment primarily affect the pointer rather than the target. ALLOCATE creates a new target but, other than breaking its connection with the specified pointer, it has no effect on the old target. Neither NULLIFY nor pointer assignment has any effect on targets. A piece of memory that was allocated and associated with a pointer will become inaccessible to a program if the pointer is nullified or associated with a different target and no other pointer was associated with this piece of memory. Such pieces of memory could be reused by the processor if it were expedient. However, whether such inaccessible memory is in fact reused is entirely processor dependent.

C.4 Clause 7 notes

C.4.1 Evaluation of function references (7.1.7)

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.4.2 Pointers in expressions (7.1.9.2)

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.4.3 Pointers in variable definition contexts (7.2.1.3, 16.6.7)

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.5 Clause 8 notes

C.5.1 The SELECT CASE construct (8.1.8)

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.5.2 Loop control (8.1.6)

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.5.3 Examples of DO constructs (8.1.6)

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:

```
      N = 0
      DO I = 1, 10
         J = I
         DO 60, K = 5, 1  ! This inner loop is never executed
            L = K
            N = N + 1
   60    CONTINUE         ! Labeled DO inside a nonlabeled DO
      END DO
```

4 After execution of the above program fragment, I = 11, J = 10, K = 5, N = 0, and L is not defined by these statements.

C.5.4 Examples of invalid DO constructs (8.1.6)

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 Clause 9 notes

C.6.1 External files (9.3)

1 This part of ISO/IEC 1539 accommodates, but does not require, file cataloging. To do this, several concepts are introduced.

C.6.1.1 File existence (9.3.2)

1 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 part of ISO/IEC 1539 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.

2 All four combinations of connection and existence can occur:

Connect	Exist	Examples
Yes	Yes	A card reader loaded and ready to be read
Yes	No	A printer before the first line is written
No	Yes	A file named 'JOAN' in the catalog
No	No	A file on a reel of tape, not known to the processor

3 Means are provided to create, delete, connect, and disconnect files.

C.6.1.2 File access (9.3.3)

1 This part of ISO/IEC 1539 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.

2 The OPEN and INQUIRE statements can be extended naturally to consider these things.

3 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.

4 Direct access to files is of a simple and commonly available type, that is, fixed-length records. The key is a positive integer.

C.6.1.3 File connection (9.5)

1 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.6.1.4 File names (9.5.6.10)

1 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 part of ISO/IEC 1539 require names to survive for any period of time longer than the execution time span of a program. Therefore, this part of ISO/IEC 1539 does not impose

cataloging as a prerequisite. The naming feature is intended to enable use of a cataloging system where one exists.

C.6.2 Nonadvancing input/output (9.3.4.2)

1 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 part of ISO/IEC 1539 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.

2 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.

3 A BACKSPACE of a file that is positioned within a record causes the specified unit to be positioned before the current record.

4 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.

5 This part of ISO/IEC 1539 provides a SIZE= specifier to be used with nonadvancing 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.

6 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.

7 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-variable*s in *io-implied-do*s 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.

8 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.

9 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.

10 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
```

11 This part of ISO/IEC 1539 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.6.3 OPEN statement (9.5.6)

1 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 (9.5.5).

2 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.)

3 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.

4 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.

5 When an OPEN statement is executed, the unit specified in the OPEN might or might not already be connected to a file. If it is already connected to a file (either through preconnection or by a prior OPEN), 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.

6 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.

7 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    '
```

8 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.  '
```

9 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
```

10 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.6.4 Connection properties (9.5.4)

1 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 (9.5.6.3).

(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 9.3.3) 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) (9.5.6.11).

(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 (9.5.6.15).

2 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 part of ISO/IEC 1539 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.6.5 Asynchronous input/output (9.6.2.5)

1 Rather than limit support for asynchronous input/output to what has been traditionally provided by facilities such as BUFFERIN/BUFFEROUT, this part of ISO/IEC 1539 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 part of ISO/IEC 1539 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 must 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.

8 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. Therefor 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.7 Clause 10 notes

C.7.1 Number of records (10.4, 10.5, 10.8.2)

1 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.

2 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

```
  PRINT 3
3 FORMAT (/)
```

will write two records that cause two blank lines if the records are printed.

C.7.2 List-directed input (10.10.3)

1 The following examples illustrate list-directed input. A blank character is represented by b.

2 **Example 1:**

Program:

```
J = 3
READ *, I
READ *, J
```

Sequential input file:

```
record 1:  b1b,4bbbbb
record 2:  ,2bbbbbbbb
```

3 Result: I = 1, J = 3.

4 Explanation: The second READ statement reads the second record. The initial comma in the record designates a null value; therefore, J is not redefined.

5 **Example 2:**

Program:

```
CHARACTER A *8, B *1
READ *, A, B
```

Sequential input file:

```
record 1:  'bbbbbbbb'
record 2:  'QXY'b'Z'
```

6 Result: A = 'bbbbbbbb', B = 'Q'

7 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.8 Clause 11 notes

C.8.1 Main program and block data program unit (11.1, 11.3)

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.8.2 Dependent compilation (11.2)

1 This part of ISO/IEC 1539, 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 part of ISO/IEC 1539 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 part of ISO/IEC 1539, 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.

2 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.

3 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.8.2.1 USE statement and dependent compilation (11.2.2)

1 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.

2 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.

3 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.

4 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.

5 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

```
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.8.2.2 Accessibility attributes (5.5.2)

1 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.8.3 Examples of the use of modules (11.2.1)

C.8.3.1 Global data (11.2.1)

1 A module could contain only data objects, for example:

```
MODULE DATA_MODULE
   SAVE
   REAL A (10), B, C (20,20)
   INTEGER :: I=0
   INTEGER, PARAMETER :: J=10
   COMPLEX D (J,J)
END MODULE DATA_MODULE
```

2 Data objects made global in this manner can have any combination of data types.

3 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
```

4 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.8.3.2 Derived types (11.2.1)

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.8.3.3 Global allocatable arrays (11.2.1)

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.8.3.4 Procedure libraries (11.2.2)

1 Interface bodies for external procedures in a library can be gathered into a module. An interface body specifies an explicit interface (12.4.2.2).

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.8.3.5 Operator extensions (11.2.2)

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.

3 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.8.3.6 Data abstraction (11.2.2)

1 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.)

2 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.

3 Here is the example module:

```
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 (-)
      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) INTERSECTION
      TYPE (SET), INTENT(IN) :: A, B
      INTERSECTION = A - (A - B)
   END FUNCTION INTERSECTION

   LOGICAL FUNCTION SUBSET (A, B)          ! Determines if set A is
      TYPE (SET), INTENT(IN) :: A, B       ! a subset of set B
      INTEGER I
      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
      INTEGER V (:)                 ! of elements and a set of elements
      INTEGER J                     ! removing duplicate elements
      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
      TYPE (SET), INTENT (IN) :: A  ! into a vector in ascending order
      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
         DO J = I + 1, A % CARD      ! A % CARD is large
            IF (VECTOR (I) > VECTOR (J)) THEN
               K = VECTOR (J); VECTOR (J) = VECTOR (I); VECTOR (I) = K
            END IF
         END DO
      END DO
   END FUNCTION VECTOR

   END MODULE INTEGER_SETS
```

4 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.8.3.7 Public entities renamed (11.2.2)

1 At times it might be necessary to rename entities that are accessed with USE statements.

2 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.8.4 Modules with submodules (11.2.3)

1 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.

2 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.

3 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.

4 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`.

5 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.

```
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
      type(color_point), allocatable :: p
    end subroutine color_point_new
  end interface

end module color_points
```

6 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.

7 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
    module subroutine inquire_palette ( pt, pal )
      use palette_stuff       ! 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

  end interface

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

end submodule color_points_b
```

9 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.

```
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                   ! just to demonstrate it's possible
    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
```

10 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.

11 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:

Plasma module				
Flow submodule			Thermal submodule	Electromagnetics submodule
Subsonic submodule	Supersonic submodule	Hypersonic submodule		

12 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.9 Clause 12 notes

C.9.1 Portability problems with external procedures (12.4.3.6)

1 There is a potential portability problem in a scoping unit that references an external procedure without explicitly declaring it to have the EXTERNAL attribute (5.5.9). 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 part of ISO/IEC 1539 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.9.2 Procedures defined by means other than Fortran (12.6.3)

1 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 part of ISO/IEC 1539. Any procedure defined by means other than Fortran is considered to be an external procedure.

2 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.9.3 Abstract interfaces and procedure pointer components (12.4, 4.5)

1 This is an example of a library module providing lists of callbacks that the user can register and invoke.

```
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.9.4 Pointers and targets as arguments (12.5.2.4, 12.5.2.6, 12.5.2.7)

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. 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 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.

3 For example, consider:

```
  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.

4 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:

```
  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.

5 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.9.5 Polymorphic Argument Association (12.5.2.9)

1 The following example illustrates the polymorphic argument association rules using the derived types defined in Note 4.56.

```
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
  CALL TUB3 ( P2 )       ! 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.9.6 Rules ensuring unambiguous generics (12.4.3.5.5)

1 The rules in 12.4.3.5.5 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.

7 For defined input/output procedures, only the `dtv` argument corresponds to something explicitly written in the program, so it is the `dtv` that is required to be distinguished. Because `dtv` arguments are required to be scalar, they cannot differ in rank. Thus this rule effectively involves only type and kind type parameters.

8 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.

9 In the special case of type-bound procedures with passed-object dummy arguments, the passed-object argument is syntactically distinguished in the reference, so rule (3) in 12.4.3.5.5 can be applied. The type of passed-object arguments is constrained in ways that prevent passed-object arguments in the same scoping unit from being type incompatible. Thus this rule effectively involves only kind type parameters and rank.

10 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):

```
INTERFACE GOOD1
  FUNCTION F1A(X)
    REAL :: F1A,X
  END FUNCTION F1A
  FUNCTION F1B(X)
    INTEGER :: F1B,X
  END FUNCTION F1B
END INTERFACE GOOD1
```

11 Whether one writes GOOD1(1.0) or GOOD1(X=1.0), the reference is to F1A because F1B would require an integer argument whereas these references provide the real constant 1.0.

12 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 rank are the distinguishing property.

13 Another common variant is the argument that satisfies (4a) and (4b) by being required in one specific and completely missing in the other:

```
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
```

14 Whether one writes GOOD2(0.0,1.0), GOOD2(0.0,Y=1.0), or GOOD2(Y=1.0,X=0.0), the reference is to F2B, because F2A has no argument in the second position or with the name 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.

15 It is possible to construct cases where different arguments are used to distinguish positionally and by name:

```
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
```

16 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 must also 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.

17 The ordering requirement in rule (4) is critical:

```
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
```

18 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.

19 Rule (1) can be used to distinguish some cases that are not covered by rule (4):

```
  INTERFACE GOOD5
    SUBROUTINE S5A(X)
      REAL :: X
    END SUBROUTINE S5A
    SUBROUTINE S5B(Y,X)
      REAL :: Y,X
    END SUBROUTINE S5B
  END INTERFACE GOOD5
```

20 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.

21 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.

22 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
```

23 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.

24 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.

25 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:

26
```
       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
```

27 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 must 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.

28 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 in-

teger 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.

29 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.10 Clause 15 notes

C.10.1 Runtime environments (15.1)

1 This part of ISO/IEC 1539 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 function) 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.10.2 Example of Fortran calling C (15.3)

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) :: RECVCOUNTS(*)
      END FUNCTION C_LIBRARY_FUNCTION
   END INTERFACE
END MODULE CLIBFUN_INTERFACE
```

1 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".

2 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.

3 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.

4 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).

5 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)
...
```

6 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.

7 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.

8 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.10.3 Example of C calling Fortran (15.3)

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.

4 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 `beta`, which is a pointer to double. An address is passed as the C actual argument.

5 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 `gamma`, which is a pointer to long. An address is passed as the C actual argument.

6 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 `delta`, which is a double array. The C actual argument is also a double array.

7 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.10.4 Example of calling C functions with noninteroperable data (15.10)

1 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

```
void ProcessBuffer(void *buffer, int n_bytes);
```

with the corresponding Fortran interface

```
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
```

2 This can be done using C_LOC if the particular Fortran processor specifies that C_LOC returns an appropriate address:

```
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.10.5 Example of opaque communication between C and Fortran (15.3)

1 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.

```
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, nonparameterized 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.10.6 Using assumed type to interoperate with C

C.10.6.1 Overview

1 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.

2 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.10.6.2 Mapping of interfaces with void * C parameters to Fortran

1 A C interface for message passing or input/output functionality could be provided in the form

```
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.

2 The first method provides a thin binding; a call to `EXAMPLE_send` from Fortran directly invokes the C function.

```
INTERFACE
   INTEGER (C_INT) FUNCTION example_send(buffer, buffer_size, handle) &
   BIND(C, NAME='EXAMPLE_send')
      USE, INTRINSIC :: ISO_C_BINDING
      TYPE(*), INTENT (IN) :: buffer(*)
      INTEGER (C_SIZE_T), VALUE :: buffer_size
      INTEGER (C_INT), INTENT (IN) :: handle
   END FUNCTION
END INTERFACE
```

3 It is assumed that this interface is declared in the specification part of the module MOD_EXAMPLE_OLD. An example of its use follows:

```
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)
```

4 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.

5 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.

```
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
```

6 It is assumed that this interface is declared in the specification part of a module MOD_EXAMPLE_NEW. Example invocations from Fortran are then

```
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.
```

7 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.10.7 Using assumed-type variables in Fortran

1 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
  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.10.8 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.10.9 Processing assumed-shape arrays in C

1 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.

2 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
```

3 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.10.10 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:

```
#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;
}

```

```c
/*
 * 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.10.11 Changing the attributes of an array

1 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;
        extents[1] = desc_alloc->dim[1].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. */
```

2 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.10.12 Creating an array section in C using CFI_section

1 The C function `set_odd` sets every second element of an array to a specific value, beginning with the first element. If does this 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.

```
        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. */
  }
```

2 The `set_odd` procedure can be called from Fortran as follows:

```
  INTEGER(C_INT) :: d(5)
  d = (/ 1, 2, 3, 4, 5 /)
  CALL set_odd(d, -1)
  PRINT *, d
```

3 This program will print something like:

```
    -1     2    -1     4    -1
```

4 During execution of the subroutine SET_ALL, its dummy argument INT_ARRAY would have size (and upper bound) 3.

5 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.

6 Following is an example C function that dynamically generates a C descriptor for an assumed-shape array and calls `set_odd`.

```
#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.10.13 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:

```
#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.

```
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.10.14 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.10.6.2. 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);
```

3 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=*), INTENT (IN) :: 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.10.6.2.

C.11 Clause 16 notes

C.11.1 Examples of host association (16.5.1.4)

1 The first two examples are examples of valid host association. The third example is an example of invalid host association.

Example 1:

```
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:

```
PROGRAM A
   TYPE T
      ...
   END 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:

```
PROGRAM Q
   REAL (KIND = 1) :: C
      ...
CONTAINS
   SUBROUTINE R
      REAL (KIND = KIND (C)) :: D  ! Invalid declaration
                                   ! See below
      REAL (KIND = 2) :: C
         ...
   END SUBROUTINE R
END PROGRAM Q
```

2 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 (7.1.12).

C.12 Array feature notes

C.12.1 Summary of features (2.4.6)

C.12.1.1 Whole array expressions and assignments (7.2.1.2, 7.2.1.3)

1 An important feature is that whole array expressions and assignments are provided. For example, in the statement

```
A = B + C * SIN (D)
```

the variables A, B, C, and D can be arrays of the same shape. It is interpreted element-by-element; that is, the sine function is taken on each element of D, each result is multiplied by the corresponding element of C, added to the corresponding element of B, and assigned to the corresponding element of A. Functions, including user-written functions, can have array results and can be generic with scalar versions. Expressions are evaluated before any assignment takes place.

C.12.1.2 Array sections (2.4.6, 6.5.3.3)

1 As well as referencing or defining a whole array, it is also possible to reference or define a subarray. For example:

```
A (:, 1:N, 2, 3:1:-1)
```

consists of a subarray containing the whole of the first dimension, positions 1 to N of the second dimension, position 2 of the third dimension and positions 1 to 3 in reverse order of the fourth dimension. This is an artificial example chosen to illustrate the different forms. One common use is to select a row or column of an array, for example:

```
A (:, J)
```

C.12.1.3 WHERE statement (7.2.3)

1 The WHERE statement applies a conforming logical array as a mask on the individual operations in the expression and in the assignment. For example:

```
WHERE (A > 0) B = LOG (A)
```

takes the logarithm only for positive components of A and makes assignments only in these positions.

2 The WHERE statement also has a block form (WHERE construct).

C.12.1.4 Automatic arrays and allocatable variables (5.2, 5.5.8.4)

1 Two features useful for writing modular software are automatic arrays, created on entry to a subprogram and destroyed on return, and allocatable variables, including arrays whose rank is fixed but whose actual size and lifetime is fully under the programmer's control through explicit ALLOCATE and DEALLOCATE statements. The declarations

```
SUBROUTINE X (N, A, B)
   REAL WORK (N, N)
   REAL, ALLOCATABLE :: HEAP (:, :)
```

specify an automatic array WORK and an allocatable array HEAP. Note that a stack is an adequate storage mechanism for the implementation of automatic arrays, but a heap will be needed for some allocatable variables.

C.12.1.5 Array constructors (4.8)

1 An array, and in particular an array constant, can be constructed with an array constructor. For example,

```
[1.0, 3.0, 7.2]
```

is a rank-one array of size 3,

```
[(1.3, 2.7, L = 1, 10), 7.1]
```

is a rank-one array of size 21 which contains the pair of real constants 1.3 and 2.7 repeated 10 times followed by 7.1, and

```
[(I, I = 1, N)]
```

is a rank-one array which contains the integers 1, 2, ..., N. Only a rank-one array can be constructed in this way, but higher dimensional arrays can be made by means of the intrinsic function RESHAPE.

C.12.2 Examples (6.5)

C.12.2.1 Unconditional array computations (6.5)

1 At the simplest level, statements such as

```
A = B + C
```

or

```
S = SUM (A)
```

can take the place of entire DO loops that would otherwise be required to perform array addition or to sum all the elements of an array.

2 Further examples of unconditional operations on arrays that are simple to write are:

matrix multiply	`P = MATMUL (Q, R)`
largest array element	`L = MAXVAL (P)`
factorial N	`F = PRODUCT ([(K, K = 2, N)])`

3 The Fourier sum $F = \sum_{i=1}^{N} a_i \times \cos x_i$ can also be computed without writing a DO loop by using the element-by-element definition of array expressions as described in Clause 7. For example,

```
F = SUM (A * COS (X))
```

The successive stages of calculation of F would then involve the arrays:

$$\begin{aligned} \mathrm{A} &= [\ \mathrm{A}\ (1), ..., \mathrm{A}\ (\mathrm{N})\ ] \\ \mathrm{X} &= [\ \mathrm{X}\ (1), ..., \mathrm{X}\ (\mathrm{N})\ ] \\ \mathrm{COS}\ (\mathrm{X}) &= [\ \mathrm{COS}\ (\mathrm{X}\ (1)), ..., \mathrm{COS}\ (\mathrm{X}\ (\mathrm{N}))\ ] \\ \mathrm{A} * \mathrm{COS}\ (\mathrm{X}) &= [\ \mathrm{A}\ (1) * \mathrm{COS}\ (\mathrm{X}\ (1)), ..., \mathrm{A}\ (\mathrm{N}) * \mathrm{COS}\ (\mathrm{X}\ (\mathrm{N}))\ ] \end{aligned}$$

4 The final scalar result is obtained simply by summing the elements of the last of these arrays. Thus, the processor is dealing with arrays at every step of the calculation.

C.12.2.2 Conditional array computations (7.2.3)

1 Suppose we wish to compute the Fourier sum in the above example, but to include only those terms $a(i)$ cos $x(i)$ that satisfy the condition that the coefficient $a(i)$ is less than 0.01 in absolute value. More precisely, we are now interested in evaluating the conditional Fourier sum $CF = \sum_{|a_i|<0.01} a_i \times \cos x_i$ where the index runs from 1 to N as before.

2 This can be done by using the MASK parameter of the SUM function, which restricts the summation of the elements of the array A * COS (X) to those elements that correspond to true elements of MASK. Clearly, the mask required is the logical array expression ABS (A) < 0.01. Note that the stages of evaluation of this expression are:

$$\begin{array}{rcl} \text{A} & = & [\ \text{A (1), ..., A (N)}\ ] \\ \text{ABS (A)} & = & [\ \text{ABS (A (1)), ..., ABS (A (N))}\ ] \\ \text{ABS (A)} < 0.01 & = & [\ \text{ABS (A (1))} < 0.01, ..., \text{ABS (A (N))} < 0.01\ ] \end{array}$$

3 The conditional Fourier sum we arrive at is

```
CF = SUM (A * COS (X), MASK = ABS (A) < 0.01)
```

4 If the mask is all false, the value of CF is zero.

5 The use of a mask to define a subset of an array is crucial to the action of the WHERE statement. Thus for example, to zero an entire array, we can write simply A = 0; but to set only the negative elements to zero, we need to write the conditional assignment

```
WHERE (A < 0)  A = 0
```

6 The WHERE statement complements ordinary array assignment by providing array assignment to any subset of an array that can be restricted by a logical expression.

7 In the Ising model described below, the WHERE statement predominates in use over the ordinary array assignment statement.

C.12.2.3 A simple program: the Ising model (6.5, 7.2.3)

C.12.2.3.1 Description of the model

1 The Ising model is a well-known Monte Carlo simulation in 3-dimensional Euclidean space which is useful in certain physical studies. We will consider in some detail how this might be programmed. The model can be described in terms of a logical array of shape N by N by N. Each gridpoint is a single logical variable which is to be interpreted as either an up-spin (true) or a down-spin (false).

2 The Ising model operates by passing through many successive states. The transition to the next state is governed by a local probabilistic process. At each transition, all gridpoints change state simultaneously. Every spin either flips to its opposite state or not according to a rule that depends only on the states of its 6 nearest neighbors in the surrounding grid. The neighbors of gridpoints on the boundary faces of the model cube are defined by assuming cubic periodicity. In effect, this extends the grid periodically by replicating it in all directions throughout space.

3 The rule states that a spin is flipped to its opposite parity for certain gridpoints where a mere 3 or fewer of the 6 nearest neighbors have the same parity as it does. Also, the flip is executed only with probability P (4), P (5), or

P (6) if as many as 4, 5, or 6 of them have the same parity as it does. (The rule seems to promote neighborhood alignments that hopefully lead to equilibrium in the long run.)

C.12.2.3.2 Problems to be solved

1 Some of the programming problems that we will need to solve in order to translate the Ising model into Fortran statements using entire arrays are

(1) counting nearest neighbors that have the same spin,

(2) providing an array function to return an array of random numbers, and

(3) determining which gridpoints are to be flipped.

C.12.2.3.3 Solutions in Fortran

1 The arrays needed are

```
LOGICAL ISING (N, N, N), FLIPS (N, N, N)
INTEGER ONES (N, N, N), COUNT (N, N, N)
REAL THRESHOLD (N, N, N)
```

and the array function needed is

```
FUNCTION RAND (N)
REAL RAND (N, N, N)
```

2 The transition probabilities are specified in the array

```
REAL P (6)
```

3 The first task is to count the number of nearest neighbors of each gridpoint g that have the same spin as g.

4 Assuming that ISING is given to us, the statements

```
ONES = 0
WHERE (ISING) ONES = 1
```

make the array ONES into an exact analog of ISING in which 1 stands for an up-spin and 0 for a down-spin.

5 The next array, COUNT, records for every gridpoint of ISING the number of spins to be found among the 6 nearest neighbors of that gridpoint. COUNT is computed by adding together 6 arrays, one for each of the 6 relative positions in which a nearest neighbor is found. Each of the 6 arrays is obtained from the ONES array by shifting the ONES array one place circularly along one of its dimensions. This use of circular shifting imparts the cubic periodicity.

```
COUNT = CSHIFT (ONES, SHIFT = -1, DIM = 1)  &
      + CSHIFT (ONES, SHIFT =  1, DIM = 1)  &
      + CSHIFT (ONES, SHIFT = -1, DIM = 2)  &
      + CSHIFT (ONES, SHIFT =  1, DIM = 2)  &
      + CSHIFT (ONES, SHIFT = -1, DIM = 3)  &
      + CSHIFT (ONES, SHIFT =  1, DIM = 3)
```

6 At this point, COUNT contains the count of nearest neighbor up-spins even at the gridpoints where the Ising model has a down-spin. It is necessary to count the down spins at the grid points, so COUNT is corrected at the down (false) points of ISING:

```
WHERE (.NOT. ISING) COUNT = 6 - COUNT
```

7 The object now is to use the counts of like-minded nearest neighbors to decide which gridpoints are to be flipped. This decision is recorded as the true elements of an array FLIPS. The decision to flip is based on the use of uniformly distributed random numbers from the interval $0 \leq p < 1$. These are provided at each gridpoint by the array function RAND. The flip occurs at a given point if and only if the random number at that point is less than a certain threshold value. In particular, making the threshold value equal to 1 at the points where there are 3 or fewer like-minded nearest neighbors guarantees that a flip occurs at those points (because p is always less than 1). Similarly, the threshold values corresponding to counts of 4, 5, and 6 are assigned P (4), P (5), and P (6) in order to achieve the desired probabilities of a flip at those points (P (4), P (5), and P (6) are input parameters in the range 0 to 1).

8 The thresholds are established by the statements:

```
THRESHOLD = 1.0
WHERE (COUNT == 4) THRESHOLD = P (4)
WHERE (COUNT == 5) THRESHOLD = P (5)
WHERE (COUNT == 6) THRESHOLD = P (6)
```

and the spins that are to be flipped are located by the statement:

```
FLIPS = RAND (N) <= THRESHOLD
```

9 All that remains to complete one transition to the next state of the ISING model is to reverse the spins in ISING wherever FLIPS is true:

```
WHERE (FLIPS)  ISING = .NOT. ISING
```

C.12.2.3.4 The complete Fortran subroutine

1 The complete code, enclosed in a subroutine that performs a sequence of transitions, is as follows:

```
SUBROUTINE TRANSITION (N, ISING, ITERATIONS, P)

   LOGICAL ISING (N, N, N), FLIPS (N, N, N)
   INTEGER ONES (N, N, N), COUNT (N, N, N)
   REAL THRESHOLD (N, N, N), P (6)

   DO I = 1, ITERATIONS
      ONES = 0
      WHERE (ISING)  ONES = 1
      COUNT = CSHIFT (ONES, -1, 1) + CSHIFT (ONES, 1, 1) &
            + CSHIFT (ONES, -1, 2) + CSHIFT (ONES, 1, 2) &
            + CSHIFT (ONES, -1, 3) + CSHIFT (ONES, 1, 3)
      WHERE (.NOT. ISING)  COUNT = 6 - COUNT
      THRESHOLD = 1.0
```

```
            WHERE (COUNT == 4)  THRESHOLD = P (4)
            WHERE (COUNT == 5)  THRESHOLD = P (5)
            WHERE (COUNT == 6)  THRESHOLD = P (6)
            FLIPS = RAND (N) <= THRESHOLD
            WHERE (FLIPS)  ISING = .NOT. ISING
         END DO

      CONTAINS
         FUNCTION RAND (N)
            REAL RAND (N, N, N)
            CALL RANDOM_NUMBER (HARVEST = RAND)
            RETURN
         END FUNCTION RAND
      END
```

C.12.2.3.5 Reduction of storage

1 The array ISING could be removed (at some loss of clarity) by representing the model in ONES all the time. The array FLIPS can be avoided by combining the two statements that use it as:

```
      WHERE (RAND (N) <= THRESHOLD)  ISING = .NOT. ISING
```

but an extra temporary array would probably be needed. Thus, the scope for saving storage while performing whole array operations is limited. If N is small, this will not matter and the use of whole array operations is likely to lead to good execution speed. If N is large, storage could be very important and adequate efficiency will probably be available by performing the operations plane by plane. The resulting code is not as elegant, but all the arrays except ISING will have size of order N^2 instead of N^3.

C.12.3 FORmula TRANslation and array processing (6.5)

C.12.3.1 General

1 Many mathematical formulas can be translated directly into Fortran by use of the array processing features.

2 We assume the following array declarations:

```
      REAL X (N), A (M, N)
```

3 Some examples of mathematical formulas and corresponding Fortran expressions follow.

C.12.3.2 A sum of products (13.7.135, 13.7.166)

1 The expression $\sum_{j=1}^{N} \prod_{i=1}^{M} a_{ij}$ can be formed using the Fortran expression

```
      SUM (PRODUCT (A, DIM=1))
```

2 The argument DIM=1 means that the product is to be computed down each column of A. If A has the value $\begin{bmatrix} \mathrm{B} & \mathrm{C} & \mathrm{D} \\ \mathrm{E} & \mathrm{F} & \mathrm{G} \end{bmatrix}$ the result of this expression is BE + CF + DG.

C.12.3.3 A product of sums (13.7.135, 13.7.166)

1 The expression $\prod_{i=1}^{M} \sum_{j=1}^{N} a_{ij}$ can be formed using the Fortran expression

```
PRODUCT (SUM (A, DIM=2))
```

2 The argument DIM = 2 means that the sum is to be computed along each row of A. If A has the value $\begin{bmatrix} \mathrm{B} & \mathrm{C} & \mathrm{D} \\ \mathrm{E} & \mathrm{F} & \mathrm{G} \end{bmatrix}$ the result of this expression is (B+C+D)*(E+F+G).

C.12.3.4 Addition of selected elements (13.7.166)

1 The expression $\sum_{x_i > 0.0} x_i$ can be formed using the Fortran expression

```
SUM (X, MASK = X>0.0)
```

2 The mask locates the positive elements of the array of rank one. If X has the vector value (0.0, −0.1, 0.2, 0.3, 0.2, −0.1, 0.0), the result of this expression is 0.7.

C.12.3.5 Sum of squared residuals (13.7.161, 13.7.166)

1 The expression $\sum_{i=1}^{N} (x_i - x_{\text{mean}})^2$ can be formed using the Fortran statements

```
XMEAN = SUM (X) / SIZE (X)
SS = SUM ((X - XMEAN) ** 2)
```

2 Thus, SS is the sum of the squared residuals.

C.12.3.6 Vector norms (13.7.2, 13.7.110, 13.7.124)

1 The L^∞-norm of vector $\mathrm{X} = (\mathrm{X}_1, \ldots, \mathrm{X}_n)$, defined as the largest of the numbers $|\mathrm{X}_1|, \ldots, |\mathrm{X}_n|$, can be formed using the Fortran expression MAXVAL (ABS (X)).

2 The L^1-norm of vector X, defined as $\sum_{i=1}^{n} |\mathrm{X}_i|$, can be formed using the Fortran expression SUM (ABS (X)).

3 The L^2-norm of vector X, defined as $\sqrt{\sum_{i=1}^{n} X_i^2}$, can be formed using the Fortran expression NORM2 (X).

C.12.3.7 Matrix norms (13.7.2, 13.7.110, 13.7.124)

1 The infinity-norm of the matrix $\mathrm{A} = \begin{bmatrix} a_{11} & \ldots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \ldots & a_{mn} \end{bmatrix}$, defined as

$$||\mathrm{A}||_\infty = \max_i \sum_{j=1}^{n} |a_{ij}|$$

can be formed using the Fortran expression MAXVAL (SUM (ABS (A), DIM = 2)).

2 The one-norm of the matrix A, defined as

$$||\mathrm{A}||_1 = \max_j \sum_{i=1}^{m} |a_{ij}|$$

can be formed using the Fortran expression MAXVAL (SUM (ABS (A), DIM = 1)).

3 There are several definitions of the two-norm of a matrix. The Frobenius or Euclidean norm of the matrix A, defined as

$$||\mathrm{A}||_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2}$$

can be formed by the Fortran expression NORM2 (A).

C.12.4 Logical queries (13.7.10, 13.7.13, 13.7.42, 13.7.110, 13.7.116 13.7.166)

1 The intrinsic functions allow quite complicated questions about tabular data to be answered without use of loops or conditional constructs. Consider, for example, the questions asked below about a simple tabulation of students' test scores.

2 Suppose the rectangular table T (M, N) contains the test scores of M students who have taken N different tests. T is an integer matrix with entries in the range 0 to 100.

3 Example: The scores on 4 tests made by 3 students are held as the table $\mathrm{T} = \begin{bmatrix} 85 & 76 & 90 & 60 \\ 71 & 45 & 50 & 80 \\ 66 & 45 & 21 & 55 \end{bmatrix}$.

4 Question: What is each student's top score?

5 Answer: MAXVAL (T, DIM = 2); in the example: [90, 80, 66].

6 Question: What is the average of all the scores?

7 Answer: SUM (T) / SIZE (T); in the example: 62.

8 Question: How many of the scores in the table are above average?

9 Answer: ABOVE = T > SUM (T) / SIZE (T); N = COUNT (ABOVE); in the example: ABOVE is the logical array (t = true, . = false): $\begin{bmatrix} \mathrm{t} & \mathrm{t} & \mathrm{t} & . \\ \mathrm{t} & . & . & \mathrm{t} \\ \mathrm{t} & . & . & . \end{bmatrix}$ and COUNT (ABOVE) is 6.

10 Question: What was the lowest score in the above-average group of scores?

11 Answer: MINVAL (T, MASK = ABOVE), where ABOVE is as defined previously; in the example: 66.

12 Question: Was there a student whose scores were all above average?

13 Answer: With ABOVE as previously defined, the answer is yes or no according as the value of the expression ANY (ALL (ABOVE, DIM = 2)) is true or false; in the example, the answer is no.

C.12.5 Parallel computations (7.1.2)

1 The most straightforward kind of parallel processing is to do the same thing at the same time to many operands. Matrix addition is a good example of this very simple form of parallel processing. Thus, the array assignment A = B + C specifies that corresponding elements of the identically-shaped arrays B and C be added together in parallel and that the resulting sums be assigned in parallel to the array A.

2 The process being done in parallel in the example of matrix addition is of course the process of addition; the array feature that implements matrix addition as a parallel process is the element-by-element evaluation of array expressions.

3 These observations lead us to look to element-by-element computation as a means of implementing other simple parallel processing algorithms.

C.12.6 Example of element-by-element computation (6.5.3)

1 Several polynomials of the same degree can be evaluated at the same point by arranging their coefficients as the rows of a matrix and applying Horner's method for polynomial evaluation to the columns of the matrix so formed.

2 The procedure is illustrated by the code to evaluate the three cubic polynomials

$$P(t) = 1 + 2t - 3t^2 + 4t^3$$
$$Q(t) = 2 - 3t + 4t^2 - 5t^3$$
$$R(t) = 3 + 4t - 5t^2 + 6t^3$$

in parallel at the point t = X and to place the resulting vector of numbers [P (X), Q (X), R (X)] in the real array RESULT (3).

3 The code to compute RESULT is just the one statement

```
RESULT = M (:, 1) + X * (M (:, 2) + X * (M (:, 3) + X * M (:, 4)))
```

where M represents the matrix M (3, 4) with value $\begin{bmatrix} 1 & 2 & -3 & 4 \\ 2 & -3 & 4 & -5 \\ 3 & 4 & -5 & 6 \end{bmatrix}$.

(Blank page)

Index

In the index, entries in *italics* denote BNF terms, and page numbers in **bold face** denote primary text or definitions.

Symbols

A

B

C

E

F

G

H

I

R

T

U

V

W

X

Z