

WD 1539-1

**J3/25-007r1
(Fortran 2028 Working Draft)**

16th July 2025 0:57

This is an internal working document of INCITS/Fortran and
ISO/IEC JTC1/SC22/WG5.

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

This page intentionally left nonblank.

Contents

Foreword	xiv
Introduction	xv
1 Scope	1
2 Normative references	2
3 Terms and definitions	3
4 Notation, conformance, and compatibility	33
4.1 Notation, symbols and abbreviated terms	33
4.1.1 Syntax rules	33
4.1.2 Constraints	34
4.1.3 Assumed syntax rules	34
4.1.4 Syntax conventions and characteristics	34
4.1.5 Text conventions	35
4.2 Conformance	35
4.3 Compatibility	36
4.3.1 Previous Fortran standards	36
4.3.2 New intrinsic procedures	36
4.3.3 Fortran 2023 compatibility	36
4.3.4 Fortran 2018 compatibility	36
4.3.5 Fortran 2008 compatibility	37
4.3.6 Fortran 2003 compatibility	39
4.3.7 Fortran 95 compatibility	40
4.3.8 Fortran 90 compatibility	40
4.3.9 FORTRAN 77 compatibility	41
4.4 Deleted and obsolescent features	41
4.4.1 General	41
4.4.2 Nature of deleted features	42
4.4.3 Nature of obsolescent features	42
5 Fortran concepts	43
5.1 High level syntax	43
5.2 Program unit concepts	46
5.2.1 Program units and scoping units	46
5.2.2 Program	47
5.2.3 Procedure	47
5.2.4 Module	47
5.2.5 Submodule	47

5.3	Execution concepts	47
5.3.1	Statement classification	47
5.3.2	Statement order	48
5.3.3	The END statement	49
5.3.4	Program execution	49
5.3.5	Execution sequence	49
5.3.6	Image execution states	50
5.3.7	Termination of execution	50
5.4	Data concepts	51
5.4.1	Type	51
5.4.2	Data value	51
5.4.3	Data entity	52
5.4.4	Definition of objects and pointers	53
5.4.5	Reference	53
5.4.6	Array	54
5.4.7	Coarray	54
5.4.8	Established coarrays	55
5.4.9	Pointer	55
5.4.10	Allocatable variables	56
5.4.11	Storage	56
5.5	Fundamental concepts	56
5.5.1	Names and designators	56
5.5.2	Statement keyword	56
5.5.3	Other keywords	56
5.5.4	Association	56
5.5.5	Intrinsic	57
5.5.6	Operator	57
5.5.7	Companion processors	57
6	Lexical tokens and source form	58
6.1	Processor character set	58
6.1.1	Characters	58
6.1.2	Letters	58
6.1.3	Digits	58
6.1.4	Underscore	58
6.1.5	Special characters	59
6.1.6	Other characters	59
6.2	Low-level syntax	59
6.2.1	Tokens	59
6.2.2	Names	59
6.2.3	Constants	60
6.2.4	Operators	60
6.2.5	Statement labels	61
6.2.6	Delimiters	62
6.3	Source form	62
6.3.1	Program units, statements, and lines	62
6.3.2	Free source form	62
6.3.3	Fixed source form (obsolescent)	64
6.4	Including source text	65

7	Types	66
7.1	Characteristics of types	66
7.1.1	The concept of type	66
7.1.2	Type classification	66
7.1.3	Set of values	66
7.1.4	Constants	66
7.1.5	Operations	66
7.2	Type parameters	67
7.3	Types, type specifiers, and values	68
7.3.1	Relationship of types and values to objects	68
7.3.2	Type specifiers	68
7.3.3	Type compatibility	70
7.4	Intrinsic types	71
7.4.1	Classification and specification	71
7.4.2	Intrinsic operations on intrinsic types	71
7.4.3	Numeric intrinsic types	71
7.4.4	Character type	75
7.4.5	Logical type	79
7.5	Derived types	79
7.5.1	Derived type concepts	79
7.5.2	Derived-type definition	80
7.5.3	Derived-type parameters	83
7.5.4	Components	85
7.5.5	Type-bound procedures	92
7.5.6	Final subroutines	94
7.5.7	Type extension	96
7.5.8	Derived-type values	98
7.5.9	Derived-type specifier	99
7.5.10	Construction of derived-type values	99
7.5.11	Derived-type operations and assignment	102
7.6	Other nonintrinsic types	102
7.6.1	Interoperable enumerations and enum types	102
7.6.2	Enumeration types	104
7.7	Binary, octal, and hexadecimal literal constants	106
7.8	Construction of array values	107
8	Attribute declarations and specifications	110
8.1	Attributes of procedures and data objects	110
8.2	Type declaration statement	110
8.3	Automatic data objects	112
8.4	Initialization	112
8.5	Attributes	113
8.5.1	Attribute specification	113
8.5.2	Accessibility attribute	113
8.5.3	ALLOCATABLE attribute	113
8.5.4	ASYNCHRONOUS attribute	114
8.5.5	BIND attribute for data entities	114
8.5.6	CODIMENSION attribute	115
8.5.7	CONTIGUOUS attribute	116

8.5.8	DIMENSION attribute	118
8.5.9	EXTERNAL attribute	122
8.5.10	INTENT attribute	122
8.5.11	INTRINSIC attribute	124
8.5.12	OPTIONAL attribute	124
8.5.13	PARAMETER attribute	124
8.5.14	POINTER attribute	125
8.5.15	PROTECTED attribute	125
8.5.16	SAVE attribute	126
8.5.17	RANK clause	126
8.5.18	TARGET attribute	127
8.5.19	VALUE attribute	127
8.5.20	VOLATILE attribute	128
8.6	Attribute specification statements	128
8.6.1	Accessibility statement	128
8.6.2	ALLOCATABLE statement	129
8.6.3	ASYNCHRONOUS statement	130
8.6.4	BIND statement	130
8.6.5	CODIMENSION statement	130
8.6.6	CONTIGUOUS statement	130
8.6.7	DATA statement	130
8.6.8	DIMENSION statement	133
8.6.9	INTENT statement	133
8.6.10	OPTIONAL statement	133
8.6.11	PARAMETER statement	134
8.6.12	POINTER statement	134
8.6.13	PROTECTED statement	134
8.6.14	SAVE statement	135
8.6.15	TARGET statement	135
8.6.16	VALUE statement	135
8.6.17	VOLATILE statement	135
8.7	IMPLICIT statement	136
8.8	IMPORT statement	138
8.9	NAMELIST statement	140
8.10	Storage association of data objects (obsolescent)	141
8.10.1	Obsolescence	141
8.10.2	EQUIVALENCE statement	141
8.10.3	COMMON statement	143
8.10.4	Restrictions on common and equivalence	145
9	Use of data objects	146
9.1	Designator	146
9.2	Variable	146
9.3	Constants	147
9.4	Scalars	147
9.4.1	Substrings	147
9.4.2	Structure components	148
9.4.3	Coindexed named objects	149
9.4.4	Complex parts	149

9.4.5	Type parameter inquiry	150
9.5	Arrays	150
9.5.1	Order of reference	150
9.5.2	Whole arrays	150
9.5.3	Array elements and array sections	150
9.5.4	Simply contiguous array designators	154
9.6	Image selectors	155
9.7	Dynamic association	156
9.7.1	ALLOCATE statement	156
9.7.2	NULLIFY statement	161
9.7.3	DEALLOCATE statement	162
9.7.4	STAT= specifier	164
9.7.5	ERRMSG= specifier	165
10	Expressions and assignment	166
10.1	Expressions	166
10.1.1	Expression semantics	166
10.1.2	Form of an expression	166
10.1.3	Precedence of operators	170
10.1.4	Evaluation of operations	171
10.1.5	Intrinsic operations	173
10.1.6	Defined operations	179
10.1.7	Evaluation of operands	181
10.1.8	Integrity of parentheses	181
10.1.9	Type, type parameters, and shape of an expression	182
10.1.10	Conformability rules for elemental operations	183
10.1.11	Specification expression	184
10.1.12	Constant expression	186
10.2	Assignment	187
10.2.1	Assignment statement	187
10.2.2	Pointer assignment	192
10.2.3	Masked array assignment – WHERE	197
10.2.4	FORALL	199
11	Execution control	203
11.1	Executable constructs containing blocks	203
11.1.1	Blocks	203
11.1.2	Rules governing blocks	203
11.1.3	ASSOCIATE construct	204
11.1.4	BLOCK construct	206
11.1.5	CHANGE TEAM construct	207
11.1.6	CRITICAL construct	209
11.1.7	DO construct	210
11.1.8	IF construct and statement	219
11.1.9	SELECT CASE construct	220
11.1.10	SELECT RANK construct	223
11.1.11	SELECT TYPE construct	225
11.1.12	EXIT statement	228
11.2	Branching	228

11.2.1	Branch concepts	228
11.2.2	GO TO statement	228
11.2.3	Computed GO TO statement	229
11.3	CONTINUE statement	229
11.4	STOP and ERROR STOP statements	229
11.5	FAIL IMAGE statement	230
11.6	NOTIFY WAIT statement	230
11.7	Image execution control	231
11.7.1	Image control statements	231
11.7.2	Segments	232
11.7.3	SYNC ALL statement	233
11.7.4	SYNC IMAGES statement	234
11.7.5	SYNC MEMORY statement	235
11.7.6	SYNC TEAM statement	236
11.7.7	EVENT POST statement	237
11.7.8	EVENT WAIT statement	237
11.7.9	FORM TEAM statement	238
11.7.10	LOCK and UNLOCK statements	239
11.7.11	STAT= and ERRMSG= specifiers in image control statements	240
12	Input/output statements	243
12.1	Input/output concepts	243
12.2	Records	243
12.2.1	Definition of a record	243
12.2.2	Formatted record	243
12.2.3	Unformatted record	244
12.2.4	Endfile record	244
12.3	External files	244
12.3.1	External file concepts	244
12.3.2	File existence	245
12.3.3	File access	245
12.3.4	File position	247
12.3.5	File storage units	249
12.4	Internal files	249
12.5	File connection	250
12.5.1	Referring to a file	250
12.5.2	Connection modes	251
12.5.3	Unit existence	252
12.5.4	Connection of a file to a unit	252
12.5.5	Preconnection	253
12.5.6	OPEN statement	253
12.5.7	CLOSE statement	258
12.6	Data transfer statements	259
12.6.1	Form of input and output statements	259
12.6.2	Control information list	260
12.6.3	Data transfer input/output list	265
12.6.4	Execution of a data transfer input/output statement	267
12.6.5	Termination of data transfer statements	278
12.7	Waiting on pending data transfer	278

12.7.1	Wait operation	278
12.7.2	WAIT statement	279
12.8	File positioning statements	280
12.8.1	Syntax	280
12.8.2	BACKSPACE statement	280
12.8.3	ENDFILE statement	281
12.8.4	REWIND statement	281
12.9	FLUSH statement	282
12.10	File inquiry statement	282
12.10.1	Forms of the INQUIRE statement	282
12.10.2	Inquiry specifiers	283
12.10.3	Inquire by output list	290
12.11	Error, end-of-record, and end-of-file conditions	290
12.11.1	Occurrence of input/output conditions	290
12.11.2	Error conditions and the ERR= specifier	291
12.11.3	End-of-file condition and the END= specifier	291
12.11.4	End-of-record condition and the EOR= specifier	291
12.11.5	IOSTAT= specifier	292
12.11.6	IOMSG= specifier	293
12.12	Restrictions on input/output statements	293
13	Input/output editing	294
13.1	Format specifications	294
13.2	Explicit format specification methods	294
13.2.1	FORMAT statement	294
13.2.2	Character format specification	294
13.3	Form of a format item list	295
13.3.1	Syntax	295
13.3.2	Edit descriptors	295
13.3.3	Fields	297
13.4	Interaction between input/output list and format	297
13.5	Positioning by format control	299
13.6	Decimal symbol	299
13.7	Data edit descriptors	299
13.7.1	Purpose of data edit descriptors	299
13.7.2	Numeric editing	300
13.7.3	Logical editing	308
13.7.4	Character editing	309
13.7.5	Generalized editing	309
13.7.6	User-defined derived-type editing	311
13.8	Control edit descriptors	311
13.8.1	Position edit descriptors	311
13.8.2	Slash editing	312
13.8.3	Colon editing	312
13.8.4	SS, SP, and S editing	313
13.8.5	LZS, LZP and LZ editing	313
13.8.6	P editing	313
13.8.7	BN and BZ editing	314
13.8.8	RU, RD, RZ, RN, RC, and RP editing	314

13.8.9	DC and DP editing	314
13.9	Character string edit descriptors	314
13.10	List-directed formatting	315
13.10.1	Purpose of list-directed formatting	315
13.10.2	Values and value separators	315
13.10.3	List-directed input	315
13.10.4	List-directed output	318
13.11	Namelist formatting	319
13.11.1	Purpose of namelist formatting	319
13.11.2	Name-value subsequences	319
13.11.3	Namelist input	320
13.11.4	Namelist output	323
14	Program units	325
14.1	Main program	325
14.2	Modules	325
14.2.1	Module syntax and semantics	325
14.2.2	The USE statement and use association	326
14.2.3	Submodules	329
14.3	Block data program units	330
15	Procedures	331
15.1	Concepts	331
15.2	Procedure classifications	331
15.2.1	Procedure classification by reference	331
15.2.2	Procedure classification by means of definition	331
15.3	Characteristics	332
15.3.1	Characteristics of procedures	332
15.3.2	Characteristics of dummy arguments	332
15.3.3	Characteristics of function results	333
15.4	Procedure interface	333
15.4.1	Interface and abstract interface	333
15.4.2	Implicit and explicit interfaces	333
15.4.3	Specification of the procedure interface	334
15.5	Procedure reference	344
15.5.1	Syntax of a procedure reference	344
15.5.2	Actual arguments, dummy arguments, and argument association	347
15.5.3	Function reference	359
15.5.4	Subroutine reference	359
15.5.5	Resolving named procedure references	360
15.5.6	Resolving type-bound procedure references	362
15.6	Procedure definition	362
15.6.1	Intrinsic procedure definition	362
15.6.2	Procedures defined by subprograms	363
15.6.3	Definition and invocation of procedures by means other than Fortran	369
15.6.4	Statement function (obsolescent)	370
15.7	Pure procedures	371
15.8	Simple procedures	373
15.9	Elemental procedures	374

15.9.1	Elemental procedure declaration and interface	374
15.9.2	Elemental function actual arguments and results	374
15.9.3	Elemental subroutine actual arguments	375
16	Intrinsic procedures and modules	376
16.1	Classes of intrinsic procedures	376
16.2	Arguments to intrinsic procedures	376
16.2.1	General rules	376
16.2.2	The shape of array arguments	377
16.2.3	Mask arguments	377
16.2.4	DIM arguments and reduction functions	377
16.3	Bit model	378
16.3.1	General	378
16.3.2	Bit sequence comparisons	378
16.3.3	Bit sequences as arguments to INT and REAL	378
16.4	Numeric models	379
16.5	Atomic subroutines	380
16.6	Collective subroutines	380
16.7	Standard generic intrinsic procedures	381
16.8	Specific names for standard intrinsic functions (obsolescent)	386
16.9	Specifications of the standard intrinsic procedures	388
16.9.1	General	388
16.10	Standard intrinsic modules	491
16.10.1	General	491
16.10.2	The ISO_FORTRAN_ENV intrinsic module	492
17	Exceptions and IEEE arithmetic	499
17.1	Overview of IEEE arithmetic support	499
17.2	Derived types, constants, and operators defined in the modules	500
17.3	The exceptions	501
17.4	The rounding modes	503
17.5	Underflow mode	504
17.6	Halting	504
17.7	The floating-point modes and status	504
17.8	Exceptional values	505
17.9	IEEE arithmetic	505
17.10	Summary of the procedures	506
17.11	Specifications of the procedures	508
17.11.1	General	508
17.12	Examples	536
18	Interoperability with C	538
18.1	General	538
18.2	The ISO_C_BINDING intrinsic module	538
18.2.1	Summary of contents	538
18.2.2	Named constants and derived types in the module	538
18.2.3	Procedures in the module	540
18.3	Interoperability between Fortran and C entities	547
18.3.1	Interoperability of intrinsic types	547

18.3.2	Interoperability with C pointer types	548
18.3.3	Interoperability of enum types	548
18.3.4	Interoperability of derived types and C structure types	548
18.3.5	Interoperability of scalar variables	550
18.3.6	Interoperability of array variables	550
18.3.7	Interoperability of procedures and procedure interfaces	551
18.4	C descriptors	553
18.5	The source file ISO_Fortran_binding.h	553
18.5.1	Summary of contents	553
18.5.2	The CFI_dim_t structure type	554
18.5.3	The CFI_cdesc_t structure type	554
18.5.4	Macros and typedefs in ISO_Fortran_binding.h	555
18.5.5	Functions declared in ISO_Fortran_binding.h	558
18.6	Restrictions on C descriptors	565
18.7	Restrictions on formal parameters	566
18.8	Restrictions on lifetimes	566
18.9	Interoperation with C global variables	567
18.9.1	General	567
18.9.2	Binding labels for common blocks and variables	568
18.10	Interoperation with C functions	568
18.10.1	Definition and reference of interoperable procedures	568
18.10.2	Binding labels for procedures	569
18.10.3	Exceptions and IEEE arithmetic procedures	569
18.10.4	Asynchronous communication	570
19	Scope, association, and definition	571
19.1	Scopes, identifiers, and entities	571
19.2	Global identifiers	571
19.3	Local identifiers	572
19.3.1	Classes of local identifiers	572
19.3.2	Local identifiers that are the same as common block names	573
19.3.3	Function results	573
19.3.4	Components, type parameters, and bindings	573
19.3.5	Argument keywords	574
19.4	Statement and construct entities	574
19.5	Association	576
19.5.1	Name association	576
19.5.2	Pointer association	579
19.5.3	Storage association (obsolescent)	582
19.5.4	Inheritance association	584
19.5.5	Establishing associations	584
19.6	Definition and undefinition of variables	585
19.6.1	Definition of objects and subobjects	585
19.6.2	Variables that are always defined	585
19.6.3	Variables that are initially defined	586
19.6.4	Variables that are initially undefined	586
19.6.5	Events that cause variables to become defined	586
19.6.6	Events that cause variables to become undefined	588
19.6.7	Variable definition context	591

19.6.8	Pointer association context	591
Annex A	(informative) Processor dependencies	592
Annex B	(informative) Deleted and obsolescent features	598
Annex C	(informative) Extended notes	602
Index		671

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 non-governmental, in liaison with ISO and IEC, also take part in the work.
- 2 The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives or www.iec.ch/members_experts/refdocs).
- 3 ISO and IEC draw attention to the possibility that the implementation of this document may involve the use of (a) patent(s). ISO and IEC take no position concerning the evidence, validity or applicability of any claimed patent rights in respect thereof. As of the date of publication of this document, ISO and IEC had not received notice of (a) patent(s) which may be required to implement this document. However, implementers are cautioned that this may not represent the latest information, which may be obtained from the patent database available at www.iso.org/patents and patents.iec.ch. ISO and IEC shall not be held responsible for identifying any or all such patent rights.
- 4 Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.
- 5 For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html. In the IEC, see www.iec.ch/understanding-standards.
- 6 This document was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.
- 7 This sixth edition cancels and replaces the fifth edition (ISO/IEC 1539-1:2023), which has been technically revised.
- 8 The main changes are as follows:
 - templates have been added;
 - other changes listed in the Introduction.
- 9 A list of all parts in the ISO/IEC 1539 series can be found on the ISO and IEC websites.
- 10 Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html and www.iec.ch/national-committees.

Introduction

- 1 This document comprises the specification of the base Fortran language, informally known as Fortran 2028. With the limitations noted in [4.3.3](#), the syntax and semantics of Fortran 2023 are contained entirely within Fortran 2028. Therefore, any standard-conforming Fortran 2023 program not affected by such limitations is a standard-conforming Fortran 2028 program. New features of Fortran 2028 can be compatibly incorporated into such Fortran 2023 programs, with any exceptions indicated in the text of this document.
- 2 Fortran 2028 contains several extensions to Fortran 2023; these are listed below.
 - Source form:
 - Data declaration:
 - Data usage and computation:
 - Input/output:
 - Execution control:
 - Intrinsic procedures:
 - Intrinsic modules:
 - Program units and procedures:
- 3 This document is organized in 19 clauses, dealing with 8 conceptual areas. These 8 areas, and the clauses in which they are treated, are:

High/low level concepts	Clauses 4 , 5 , 6
Data concepts	Clauses 7 , 8 , 9
Computations	Clauses 10 , 16 , 17
Execution control	Clause 11
Input/output	Clauses 12 , 13
Program units	Clauses 14 , 15
Interoperability with C	Clause 18
Scoping and association rules	Clause 19

- 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 Scope

- 1 This document specifies the form and establishes the interpretation of programs expressed in the base Fortran language. The purpose of this document is to promote portability, reliability, maintainability, and efficient execution of Fortran programs for use on a variety of computing systems.
- 2 This document specifies
 - the forms that a program written in the Fortran language can take,
 - the rules for interpreting the meaning of a program and its data,
 - the form of the input data to be processed by such a program, and
 - the form of the output data resulting from the use of such a program.
- 3 Except where stated otherwise, requirements and prohibitions specified by this document apply to programs rather than processors.
- 4 This document does not specify
 - the mechanism by which programs are transformed for use on computing systems,
 - the operations required for setup and control of the use of programs on computing systems,
 - the method of transcription of programs or their input or output data to or from a storage medium,
 - the program and processor behavior when this document fails to establish an interpretation except for the processor detection and reporting requirements in items (2) to (10) of 4.2,
 - the maximum number of [images](#), or the size or complexity of a program and its data that will exceed the capacity of any particular computing system or the capability of a particular processor,
 - the mechanism for determining the number of [images](#) of a program,
 - the physical properties of an [image](#) or the relationship between [images](#) and the computational elements of a computing system,
 - the physical properties of the representation of quantities and the method of rounding, approximating, or computing numeric values on a particular processor, except by reference to ISO/IEC 60559:2020 under conditions specified in Clause 17,
 - the physical properties of input/output records, files, and units, or
 - the physical properties and implementation of storage.

2 Normative references

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

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

ISO/IEC 9899:2018, *Programming languages—C*

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

ISO/IEC/IEEE 60559:2020, *Information technology — Microprocessor Systems — Floating-Point arithmetic*

3 Terms and definitions

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

ISO and IEC maintain terminology databases for use in standardization at the following addresses:

- ISO Online browsing platform: available at <https://www.iso.org/obp>
- IEC Electropedia: available at <https://www.electropedia.org/>

3.1

actual argument

entity that determines argument association

Note 1 to entry: See [15.5.2.3](#) and [15.5.2.4](#).

Note 2 to entry: An *actual-arg*, *consequent-arg*, or *variable* in a *defined assignment statement*, are all examples of actual arguments.

3.2

allocatable

having the *ALLOCATABLE* attribute

Note 1 to entry: See [8.5.3](#).

3.3

array

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

Note 1 to entry: See [8.5.8](#) and [9.5](#).

3.3.1

array element

scalar *subobject* ([3.138](#)) of an array that has the same type and type parameters as the array

Note 1 to entry: Array elements are described in [9.5.3](#).

3.3.2

array pointer

array with the *POINTER* attribute

Note 1 to entry: The *POINTER* attribute is described in [8.5.14](#).

3.3.3

array section

array *subobject* ([3.138](#)) designated by *array-section*, and which is itself an array

Note 1 to entry: Array sections are described in [9.5.3.4](#).

3.3.4**assumed-shape array**

nonallocatable nonpointer *dummy argument* (3.59) array that takes its shape from its *effective argument* (3.60)

Note 1 to entry: Assumed-shape arrays are described in 8.5.8.3.

3.3.5**assumed-size array**

dummy argument (3.59) array whose size is assumed from that of its *effective argument* (3.60)

Note 1 to entry: Assumed-size arrays are described in 8.5.8.5.

3.3.6**deferred-shape array**

allocatable (3.2) array or *array pointer* (3.3.2)

Note 1 to entry: Deferred-shape arrays are described in 8.5.8.4.

3.3.7**explicit-shape array**

array declared with an *explicit-shape-spec-list* or *explicit-shape-bounds-spec*, which specifies explicit values for the *bounds* (3.17) in each dimension of the array

Note 1 to entry: Explicit-shape arrays are described in 8.5.8.2.

3.4**ASCII character**

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

3.5**associate name**

name of *construct entity* (3.35) associated with a selector of an *ASSOCIATE*, *CHANGE TEAM*, *SELECT RANK*, or *SELECT TYPE* construct

Note 1 to entry: See 11.1.3, 11.1.5, 11.1.10, and 11.1.11.

3.6**associating entity**

entity that did not exist prior to a dynamically-established association

Note 1 to entry: Dynamically-established associations are described in 19.5.5.

3.7**association**

inheritance association (3.7.4), *name association* (3.7.6), *pointer association* (3.7.7), or *storage association* (3.7.8)

3.7.1**argument association**

association between an *effective argument* (3.60) and a *dummy argument* (3.59)

Note 1 to entry: Argument association is described in 15.5.2.

3.7.2**construct association**

association between a selector and an *associate name* (3.5) in an *ASSOCIATE*, *CHANGE TEAM*, *SELECT RANK*, or *SELECT TYPE* construct

Note 1 to entry: See 11.1.3, 11.1.5, 11.1.10, 11.1.11, and 19.5.1.6.

3.7.3**host association**

name association, other than argument association, between entities in a submodule or contained *scoping unit* (3.120) and entities in its host

Note 1 to entry: Host association is described in 19.5.1.4.

3.7.4**inheritance association**

association between the inherited components of an *extended type* (3.144.5) and the components of its *parent component* (3.30.2)

Note 1 to entry: Inheritance association is described in 19.5.4.

3.7.5**linkage association**

association between a variable or common block with the *BIND attribute* and a C global variable

Note 1 to entry: Linkage association is described in 18.9 and 19.5.1.5.

Note 2 to entry: Common blocks are obsolescent.

3.7.6**name association**

argument association (3.7.1), *construct association* (3.7.2), *host association* (3.7.3), *linkage association* (3.7.5), or *use association* (3.7.9)

Note 1 to entry: Name association is described in 19.5.1.

3.7.7**pointer association**

association between a *pointer* (3.104) and a procedure or a variable with the *TARGET attribute*

Note 1 to entry: Pointer association is described in 19.5.2.

3.7.8**storage association**

association between storage sequences

Note 1 to entry: Storage association is described in 19.5.3.

3.7.9**use association**

association between entities in a module and entities in a scoping unit or construct that references that module, as specified by a *USE statement*

Note 1 to entry: Use association is described in 14.2.2.

3.8**assumed-rank dummy data object**

dummy data object (3.59.1) that assumes the rank, shape, and size of its *effective argument* (3.60)

Note 1 to entry: Assumed-rank entities are described in 8.5.8.7.

3.9**assumed-type**

declared with a TYPE(*) type specifier

Note 1 to entry: See 7.3.2.

3.10**attribute**

property of an entity that determines its uses

Note 1 to entry: Attributes of procedures and data objects are described in 8.1.

3.11**automatic data object**

nondummy *data object* (3.42) with a *type parameter* (3.144.12) or *array bound* (3.17) that depends on the value of a *specification expression* (3.128) that is not a *constant expression* (3.36)

Note 1 to entry: Automatic data objects are described in 8.3.

3.12**base object**

object designated by the leftmost *part-name*

Note 1 to entry: Base objects are described in 9.4.2.

Note 2 to entry: This only applies to the *data-ref* syntax (R911).

3.13**binding**

type-bound procedure (3.109.7) or *final subroutine* (3.69)

3.14**binding name**

name given to a specific or generic *type-bound procedure* (3.109.7) in the type definition

Note 1 to entry: Type-bound procedures are described in 7.5.5.

3.15**binding label**

default character value specifying the name by which a global entity with the **BIND attribute** is known to the *companion processor* (3.29)

Note 1 to entry: Binding labels are described in 18.10.2 and 18.9.2.

3.16**block**

sequence of executable constructs formed by the syntactic class *block*

Note 1 to entry: A block is treated as a unit by the executable constructs described in 11.1.

3.17**bound****array bound**

limit of a dimension of an *array* (3.3)

3.18**branch target statement**

statement whose *statement label* (3.132) appears as a *label* in a GO TO statement, computed GO TO statement, *alt-return-spec*, END= specifier, EOR= specifier, or ERR= specifier

Note 1 to entry: A branch target statement shall be an *action-stmt*, *associate-stmt*, *end-associate-stmt*, *if-then-stmt*, *end-if-stmt*, *select-case-stmt*, *end-select-stmt*, *select-rank-stmt*, *end-select-rank-stmt*, *select-type-stmt*, *end-select-type-stmt*, *do-stmt*, *end-do-stmt*, *block-stmt*, *end-block-stmt*, *critical-stmt*, *end-critical-stmt*, *forall-construct-stmt*, *where-construct-stmt*, *end-function-stmt*, *end-mp-subprogram-stmt*, *end-program-stmt*, or *end-subroutine-stmt*. Branching is described in 11.2.1.

Note 2 to entry: Computed GO TO statements and alternate return specifiers are obsolescent.

3.19**C address**

value of type C_PTR or C_FUNPTR from the intrinsic module ISO_C_BINDING identifying a location

Note 1 to entry: This is the concept that ISO/IEC 9899:2018 calls the address. This can apply to a variable or procedure.

3.20**C descriptor**

C structure of type CFI_cdesc_t defined in the source file ISO_Fortran_binding.h

Note 1 to entry: C descriptors and the source file ISO_Fortran_binding.h are described in 18.4 and 18.5.

3.21**character context**

within a character literal constant or within a character string edit descriptor

Note 1 to entry: Character literal constants are described in 7.4.4. Character string edit descriptors are described in 13.3.2.

3.22**characteristics**

properties used to determine compatibility or consistency

Note 1 to entry: A *dummy argument* (3.59) has the characteristic of being a *dummy data object* (3.59.1), *dummy procedure* (3.109.1), or an asterisk (alternate return indicator; these are obsolescent). A *dummy data object* (3.59.1) has the additional characteristics listed in 15.3.2.2. A *dummy procedure* (3.109.1) has the additional characteristics listed in 15.3.2.3.

A function result has the characteristics listed in 15.3.3. A procedure has the characteristics listed in 15.3.1. The characteristics of intrinsic procedures are listed in 16.9.

3.23**coarray**

component (3.30), or *variable* (3.151), that has nonzero *corank* (3.39)

3.23.1**established coarray**

coarray (3.23) that is accessible using an *image-selector*

Note 1 to entry: Established coarrays are described in 5.4.8.

3.24**cobound**

bound (limit) of a *codimension* (3.25)

3.25**codimension**

dimension of the pattern formed by a set of corresponding *coarrays* (3.23)

3.26**coindexed object**

data object (3.42) whose *designator* (3.56) includes an *image-selector*

Note 1 to entry: Image selectors are described in 9.6.

3.27**collating sequence**

one-to-one mapping from a character set into the nonnegative integers

Note 1 to entry: Collating sequences are described in 7.4.4.4.

3.28**common block**

block of physical storage specified by a **COMMON statement**

Note 1 to entry: Common blocks are described in 8.10.3.

Note 2 to entry: Common blocks are obsolescent.

3.28.1**blank common**

unnamed common block

3.29**companion processor**

processor-dependent mechanism by which global data and procedures can be referenced or defined

Note 1 to entry: Companion processors are described in 5.5.7.

3.30**component**

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

Note 1 to entry: Components are described in 7.5.4.

3.30.1**direct component**

one of the components, or one of the direct components of a nonpointer nonallocatable component

Note 1 to entry: See 7.5.1.

3.30.2**parent component**

component of an *extended type* (3.144.5) whose type is that of the *parent type* (3.144.10) and whose components are *inheritance associated* (3.7.4) with the *inherited* (3.84) components of the parent type

Note 1 to entry: Inheritance and the parent component are described in 7.5.7.2.

3.30.3**potential subobject component**

nonpointer component, or potential subobject component of a nonpointer component

Note 1 to entry: See 7.5.1.

3.30.4**subcomponent**

⟨*structure* (3.136)⟩ *direct component* (3.30.1) that is a *subobject* (3.138) of the structure

Note 1 to entry: See 9.4.2.

3.30.5**ultimate component**

component that is of *intrinsic type* (3.144.8), a *pointer* (3.104), or *allocatable* (3.2); or an ultimate component of a nonpointer nonallocatable component of derived type

3.31**component order**

ordering of the nonparent components of a derived type that is used for *intrinsic* (3.90) formatted input/output and, where component keywords are not used, *structure constructors* (3.136.2)

Note 1 to entry: Component order is described in 7.5.4.7.

3.32**conformable**

having the same shape, or one being an array and the other being scalar

Note 1 to entry: This is a relationship between two data entities.

3.33**connected**

relationship between a *unit* (3.148) and a file: each is connected if and only if the unit refers to the file

Note 1 to entry: See 12.5.4.

3.34**constant**

data object (3.42) that has a value and which cannot be defined, redefined, or become undefined during execution of a program

Note 1 to entry: See 6.2.3 and 9.3.

3.34.1**literal constant**

constant that does not have a name

Note 1 to entry: A literal constant has the syntax *literal-constant* (R605), and is of intrinsic type (7.4).

3.34.2**named constant**

named *data object* (3.42) with the **PARAMETER** attribute

3.35**construct entity**

entity whose identifier has the scope of a construct

Note 1 to entry: The scoping of such entities is described in 19.1 and 19.4.

3.36**constant expression**

expression satisfying requirements that ensure its value is constant

Note 1 to entry: A constant expression shall satisfy the requirements in 10.1.12.

3.37**contiguous**

⟨array⟩ whose array elements, in order, are not separated by other data objects

Note 1 to entry: The requirements for contiguous are defined in 8.5.7.

3.38**contiguous**

⟨multi-part data object⟩ whose parts, in order, are not separated by other data objects

3.39**corank**

number of *codimensions* (3.25) of a *coarray* (3.23), or zero for objects that are not coarrays

Note 1 to entry: See 5.4.7 and 8.5.6.

3.40**cosubscript**

scalar integer expression in an image selector

Note 1 to entry: The syntax of an image selector is specified by the BNF rule *image-selector*(R926). The syntax of a cosubscript is specified by the BNF rule *cosubscript*(R927).

3.41**data entity**

data object (3.42), result of the evaluation of an expression, or the result of the execution of a function reference

3.42**data object****object**

constant, variable, or subobject of a constant

Note 1 to entry: See 7.1.4, 9.2, and 5.4.3.2.4.

3.43**decimal symbol**

character that separates the whole and fractional parts in the decimal representation of a real number in a file

Note 1 to entry: See 13.6.

3.44**declaration**

specification of attributes for various program entities Note 1 to entry: Often this involves specifying the type of a named data object or specifying the shape of a named array object.

3.45**default initialization**

mechanism for automatically initializing pointer components to have a defined pointer association status, and nonpointer components to have a particular value

Note 1 to entry: Default initialization is described in 7.5.4.6.

3.46**default-initialized**

⟨*subcomponent* (3.30.4)⟩ subject to a *default initialization* (3.45) specified in the type definition for that component

3.47**definable**

capable of *definition* (3.53) and permitted to become *defined* (3.48)

3.48**defined**

⟨*data object* (3.42)⟩ with a valid value

3.49**defined**

⟨*pointer* (3.104)⟩ whose pointer association status is associated or disassociated

Note 1 to entry: Pointer association is described in 19.5.2.2.

3.50**defined assignment**

assignment defined by a procedure

Note 1 to entry: See 10.2.1.4 and 15.4.3.4.3.

3.51**defined input/output**

input/output defined by a procedure and accessed via a *defined-io-generic-spec*

Note 1 to entry: See syntax rule R1509, and 12.6.4.8.

3.52**defined operation**

operation defined by a procedure

Note 1 to entry: See 10.1.6.1 and 15.4.3.4.2.

3.53**definition**

⟨*data object* (3.42)⟩ process by which the data object becomes defined

Note 1 to entry: Such events are listed in 19.6.5.

3.54**definition**

⟨derived type, interoperable enumeration, enumeration type, or procedure⟩ specification of the type, enumeration, or procedure

Note 1 to entry: See 7.5.2, 7.6.1, 7.6.2, and 15.6.

3.55**descendant**

submodule that extends a module or submodule, or that extends another descendant thereof

Note 1 to entry: This is a relationship between a module or submodule and a submodule. Submodules are described in 14.2.3.

3.56**designator**

name followed by zero or more component selectors, complex part selectors, array section selectors, array element selectors, image selectors, and substring selectors

Note 1 to entry: Designators are defined in 9.1.

3.56.1**complex part designator**

designator that designates the real or imaginary part of a complex *data object* (3.42), independently of the other part

Note 1 to entry: Complex parts are described in 9.4.4.

3.56.2**object designator****data object designator**

designator (3.56) for a *data object* (3.42)

Note 1 to entry: An object name is a special case of an object designator.

3.56.3**procedure designator**

designator (3.56) for a procedure

3.57**disassociated**

⟨pointer association⟩ pointer association status of not being associated with any target and not being undefined

Note 1 to entry: Pointer association status is described in 19.5.2.2.

3.58**disassociated**

⟨pointer⟩ whose pointer association status is disassociated

3.59**dummy argument**

entity whose identifier appears in a *dummy-arg-list* or *dummy-arg-name-list* in a **FUNCTION**, **SUBROUTINE**, **ENTRY**, or **statement function** statement, or whose name can be used as an *argument keyword* (3.94.1) in a reference to an *intrinsic* (3.90) procedure or a procedure in an intrinsic module

Note 1 to entry: The **ENTRY** statement is obsolescent.

3.59.1**dummy data object**

dummy argument (3.59) that is a data object

3.59.2**dummy function**

dummy procedure (3.109.1) that is a function

3.60**effective argument**

entity that is argument-associated with a *dummy argument* (3.59)

Note 1 to entry: Argument association is described in 15.5.2.4.

3.61**effective item**

scalar object treated as a single entity in input/output

Note 1 to entry: An effective item results from the application of the rules in [12.6.3](#) to an input/output list.

3.62**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 to entry: Combination of scalar and array operands or arguments combine the scalar operand(s) with each element of the array operand(s).

3.62.1**elemental assignment**

assignment that operates elementally

3.62.2**elemental operation**

operation that operates elementally

3.62.3**elemental operator**

operator in an elemental operation

3.62.4**elemental procedure**

procedure that can be used elementally

Note 1 to entry: User-defined elemental procedures are described in [15.9](#).

3.62.5**elemental reference**

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

3.62.6**elemental subprogram**

subprogram with the [ELEMENTAL](#) prefix

Note 1 to entry: See [15.9.1](#).

3.63**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](#)

Note 1 to entry: The [end-block-data-stmt](#) is obsolescent.

3.64**explicit initialization**

initialization of a data object by a specification statement

Note 1 to entry: See [8.4](#) and [8.6.7](#).

3.65**extent**

number of elements in a single dimension of an *array* (3.3)

3.66**external file**

file that exists in a medium external to the program

Note 1 to entry: See 12.3.

3.67**external unit****external input/output unit**

entity that can be *connected* (3.33) to an *external file* (3.66)

Note 1 to entry: External units and their connection are described in 12.5.3 and 12.5.4.

3.68**file storage unit**

unit of storage in a *stream file* (3.135) or an unformatted *record file* (3.116)

Note 1 to entry: File storage units are described in 12.3.5.

3.69**final subroutine**

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

Note 1 to entry: See 7.5.6.

3.70**finalizable**

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

3.71**finalizable**

⟨nonpointer data entity⟩ of finalizable type

3.72**finalization**

process of calling final subroutines when certain events occur

Note 1 to entry: These events are listed in 7.5.6.3.

3.73**function**

procedure that is invoked by an expression

3.74**function result**

entity that returns the value of a function

Note 1 to entry: See 15.6.2.2.

3.75**generic identifier**

sequence of tokens that identifies a generic set of procedures or operations

Note 1 to entry: See [15.4.3.4](#). In this context, an operation could be defined input/output or an assignment.

3.76**host instance**

instance of the host procedure that supplies the host environment

Note 1 to entry: Instances are described in [15.6.2.4](#).

Note 2 to entry: This is only applicable to an *internal procedure* ([3.109.3](#)), or a *dummy procedure* ([3.109.1](#)) or *procedure pointer* ([3.104.2](#)) that is associated with an *internal procedure* ([3.109.3](#)).

3.77**host scoping unit****host**

scoping unit ([3.120](#)) immediately surrounding another scoping unit, or the scoping unit extended by a sub-module

3.78**IEEE infinity**

ISO/IEC/IEEE 60559:2020 conformant infinite floating-point value

3.79**IEEE NaN****NaN**

ISO/IEC/IEEE 60559:2020 conformant floating-point datum that does not represent a number

3.80**image**

instance of a Fortran program

Note 1 to entry: See [5.3.4](#).

3.80.1**active image**

image ([3.80](#)) that has not failed or stopped

Note 1 to entry: Image execution states are described in [5.3.6](#).

3.80.2**failed image**

image ([3.80](#)) that has not initiated termination but which has ceased to participate in program execution

3.80.3**stopped image**

image ([3.80](#)) that has initiated *normal termination*

3.81**image index**

integer value identifying an *image* ([3.80](#)) within a *team* ([3.142](#))

3.82**image control statement**

statement that affects the execution ordering between *images* (3.80)

Note 1 to entry: Image execution control is described in 11.7.

3.83**inclusive scope**

nonblock *scoping unit* (3.120) plus every *block scoping unit* (3.120.1) whose *host* (3.77) is that scoping unit or that is nested within such a block scoping unit

Note 1 to entry: That is, inclusive scope is the scope as if **BLOCK constructs** were not scoping units.

3.84**inherit**

acquire entities (*components* (3.30), *type-bound procedures* (3.109.7), and *type parameters* (3.144.12)) through type extension from the parent type

Note 1 to entry: Inheritance is described in 7.5.7.2.

3.85**inquiry function**

intrinsic (3.90) function, or function in an intrinsic module, whose result depends on the properties of one or more of its arguments instead of their values

3.86**interface**

⟨procedure⟩ name, procedure characteristics, dummy argument names, binding label, and generic identifiers

Note 1 to entry: See 15.4.1.

3.86.1**abstract interface**

set of procedure characteristics with dummy argument names

3.86.2**explicit interface**

interface of a procedure that includes all the characteristics of the procedure and names for its dummy arguments except for asterisk dummy arguments

Note 1 to entry: See 15.4.2.

Note 2 to entry: Asterisk dummy arguments (alternate return indicators) are obsolescent.

3.86.3**generic interface**

set of procedure interfaces identified by a *generic identifier* (3.75)

3.86.4**implicit interface**

interface of a procedure that is not an explicit interface

Note 1 to entry: See 15.4.2 and 15.4.3.8.

3.86.5**specific interface**

interface (3.86) identified by a nongeneric name

3.87**interface block**

abstract interface block (3.87.1), *generic interface block* (3.87.2), or *specific interface block* (3.87.3)

Note 1 to entry: Interface blocks are described in 15.4.3.2.

3.87.1**abstract interface block**

interface block with the **ABSTRACT** keyword; collection of *interface bodies* that specify named *abstract interfaces* (3.86.1)

3.87.2**generic interface block**

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

3.87.3**specific interface block**

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

3.88**interoperable**

⟨Fortran entity⟩ equivalent to an entity defined by or definable by the *companion processor* (3.29)

Note 1 to entry: Interoperability between Fortran and C entities is described in 18.3.

3.89**interoperable**

⟨C entity⟩ equivalent to an entity defined by or definable by the Fortran processor

3.90**intrinsic**

type, procedure, module, assignment, operator, or input/output operation defined in this document and accessible without further definition or specification, or a procedure or module provided by a processor but not defined in this document

3.90.1**standard intrinsic**

intrinsic, defined in this document

3.91**internal file**

character variable that is *connected* (3.33) to an *internal unit* (3.92)

Note 1 to entry: Internal files are described in 12.4. File connection is described in 12.5.4.

3.92**internal unit**

input/output unit (3.148) that is *connected* (3.33) to an *internal file* (3.91)

3.93**ISO 10646 character**

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

3.94**keyword**

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

3.94.1**argument keyword**

word that identifies the corresponding *dummy argument* (3.59) in an actual argument list

Note 1 to entry: Argument correspondence is described in 15.5.2.1.

3.94.2**component keyword**

word that identifies a *component* (3.30) in a *structure constructor* (3.136.2)

3.94.3**statement keyword**

word that is part of the syntax of a statement

Note 1 to entry: Statement keywords are described in 5.5.2.

3.94.4**type parameter keyword**

word that identifies a *type parameter* (3.144.12) in a *type-param-spec*

3.95**lexical token**

keyword, name, literal constant other than a complex literal constant, operator, label, delimiter, comma, =, =>, :, ::, :::, or %

Note 1 to entry: See 6.2.

3.96**line**

sequence of zero or more characters

3.97**main program**

program unit (3.113) that is not a *subprogram* (3.139), *module* (3.99), *submodule* (3.137), or *block data program unit*

Note 1 to entry: See 14.1.

3.98**masked array assignment**

[assignment statement](#) in a [WHERE statement](#) or [WHERE construct](#)

Note 1 to entry: See [10.2.3](#).

3.99**module**

[program unit](#) ([3.113](#)) that can contain, or access from another module, definitions that can be made accessible to other program units

Note 1 to entry: Modules are described in [14.2](#).

3.100**name**

identifier of a program constituent, beginning with an alphabetic character and containing only alphanumeric characters and underscores

Note 1 to entry: The form of a name follows the rules given in [6.2.2](#).

3.101**operand**

data value that is the subject of an operator

3.102**operator**

[intrinsic-operator](#), [defined-unary-op](#), or [defined-binary-op](#)

Note 1 to entry: These are defined by the syntax rules R608, R1004, and R1024.

3.103**passed-object dummy argument**

dummy argument of a [type-bound procedure](#) ([3.109.7](#)) or [procedure pointer](#) ([3.104.2](#)) component that becomes associated with the object through which the procedure is invoked

Note 1 to entry: This is described in [7.5.4.5](#).

3.104**pointer**

[data pointer](#) ([3.104.1](#)) or [procedure pointer](#) ([3.104.2](#))

3.104.1**data pointer**

[data entity](#) ([3.41](#)) with the [POINTER](#) attribute

Note 1 to entry: See [8.5.14](#).

3.104.2**procedure pointer**

procedure with the [POINTER](#) attribute

3.104.3**local procedure pointer**

procedure pointer (3.104.2) that is part of a *local variable* (3.151.2), or a named procedure pointer that is not a *dummy argument* (3.59) or accessed by use or host association

3.105**pointer assignment**

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

Note 1 to entry: The pointer assignment statement is described in 10.2.2. Derived-type intrinsic assignment is described in 10.2.1.2.

3.106**polymorphic**

⟨data entity⟩ able to be of differing *dynamic types* (3.144.4) during program execution

Note 1 to entry: Polymorphic data objects are declared with the CLASS type specifier (7.3.2.3).

3.107**polymorphic**

⟨function⟩ having a result that is a polymorphic data entity

3.108**preconnected**

connected (3.33) at the beginning of execution of the program

Note 1 to entry: Preconnection is described in 12.5.5.

3.109**procedure**

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

3.109.1**dummy procedure**

dummy argument (3.59) that is a procedure

Note 1 to entry: See 15.2.2.3.

3.109.2**external procedure**

procedure defined by an *external subprogram* (3.139.1) or by means other than Fortran

Note 1 to entry: The syntax of an external subprogram is defined by R503. See also 15.6.3.

3.109.3**internal procedure**

procedure defined by an *internal subprogram* (3.139.2)

Note 1 to entry: The syntax of an internal subprogram is defined by R512.

3.109.4**module procedure**

procedure defined by a module subprogram, or a specific procedure provided by an intrinsic module

Note 1 to entry: The syntax of a module subprogram is defined by R1408.

3.109.5**pure procedure**

procedure declared or defined to be pure (15.7)

3.109.6**simple procedure**

procedure declared or defined to be simple

Note 1 to entry: Simple procedures are described in 15.8.

3.109.7**type-bound procedure**

procedure that is bound to a derived type and referenced via an object of that type

Note 1 to entry: Type-bound procedures are described in 7.5.5.

3.110**processor**

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

3.111**processor dependent**

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

Note 1 to entry: For example, the number of decimal digits displayed in list-directed output of a real value may vary across processors.

3.112**program**

set of Fortran *program units* (3.113) and entities defined by means other than Fortran that includes exactly one *main program* (3.97)

3.113**program unit**

main program (3.97), *external subprogram* (3.139.1), *module* (3.99), *submodule* (3.137), or *block data program unit*

Note 1 to entry: See 5.2.1.

Note 2 to entry: The block data program unit is obsolescent.

3.114**rank**

number of array dimensions of a *data entity* (3.41) that is an array, or zero for a scalar entity

3.115**record**

sequence of values or characters in a file

Note 1 to entry: See [12.2](#).

3.116**record file**

file composed of a sequence of records

Note 1 to entry: See [12.1](#).

3.117**reference**

[data object reference](#) (3.117.1), [procedure reference](#) (3.117.4), or [module reference](#) (3.117.3)

3.117.1**data object reference**

appearance of a [data object designator](#) (3.56.2) in a context requiring its value at that point during execution

3.117.2**function reference**

appearance of the [procedure designator](#) (3.56.3) for a function, or operator symbol for a [defined operation](#) (3.52), in a context requiring execution of the function during expression evaluation

Note 1 to entry: See [15.5.3](#).

3.117.3**module reference**

appearance of a module name in a [USE statement](#)

Note 1 to entry: See [14.2.2](#).

3.117.4**procedure reference**

appearance of a [procedure designator](#) (3.56.3), operator symbol, or assignment symbol in a context requiring execution of the procedure at that point during execution; or occurrence of defined input/output or derived-type [finalization](#) (3.72)

Note 1 to entry: Defined input/output is described in [12.6.4.8](#).

3.118**saved**

having the [SAVE attribute](#)

Note 1 to entry: The SAVE attribute is described in [8.5.16](#).

3.119**scalar**

[data entity](#) (3.41) that can be represented by a single value of the type and that is not an [array](#) (3.3)

3.120**scoping unit**

[BLOCK construct](#), derived-type definition, [interface body](#), [program unit](#) (3.113), or subprogram, excluding all nested scoping units in it

3.120.1**block scoping unit**

scoping unit of a [BLOCK construct](#)

3.121**segment**

maximal sequence of executions on an [image](#) (3.80) of statements other than [image control statements](#) (3.82)

Note 1 to entry: Segments are described in [11.7.2](#).

3.122**sequence**

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

3.123**sequence structure**

scalar [data object](#) (3.42) of a [sequence type](#) (3.124)

3.124**sequence type**

derived type with the [SEQUENCE attribute](#)

Note 1 to entry: Sequence types are described in [7.5.2.3](#).

3.124.1**character sequence type**

sequence type with no [allocatable](#) (3.2) or [pointer](#) (3.104) [components](#) (3.30), and whose components are all default character or of another character sequence type

3.124.2**numeric sequence type**

sequence type with no [allocatable](#) (3.2) or [pointer](#) (3.104) [components](#) (3.30), and whose components are all default complex, default integer, default logical, default real, double precision real, or of another numeric sequence type

3.125**shape**

array dimensionality of a data entity, represented as a rank-one array whose size is the [rank](#) (3.114) of the data entity and whose elements are the extents of the data entity

Note 1 to entry: Thus the shape of a scalar data entity is an array with rank one and size zero.

3.126**simply contiguous**

satisfying requirements that ensure it is contiguous

Note 1 to entry: An entity that is simply contiguous shall satisfy the requirements specified in 9.5.4.

Note 2 to entry: These requirements are simple ones which make it clear that a designator or variable designates a *contiguous* (3.37) array. Only an array designator or variable can be simply contiguous.

3.127**size**

⟨array⟩ total number of elements in the *array* (3.3)

3.128**specification expression**

expression satisfying requirements that make it suitable for use in specifications

Note 1 to entry: A specification expression shall satisfy the requirements specified in 10.1.11.

3.128.1**component specification expression**

specification expression satisfying additional requirements that make it suitable for use in specifications in a *component definition statement*

Note 1 to entry: A component specification expression shall specify the additional requirements specified in 10.1.11.

3.129**specific name**

name that is not a generic name

3.130**statement**

sequence of one or more complete or partial lines satisfying a syntax rule that ends in *-stmt*

Note 1 to entry: See 6.3.

3.130.1**executable statement**

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

3.130.2**nonexecutable statement**

statement that is not an *executable statement* (3.130.1)

3.131**statement entity**

entity whose identifier has the scope of a statement or part of a statement

Note 1 to entry: See 19.1 and 19.4.

3.132**statement label****label**

unsigned positive number of up to five digits that refers to an individual statement

Note 1 to entry: Statement labels are described in [6.2.5](#).

3.133**storage sequence**

contiguous sequence of *storage units* ([3.134](#))

3.134**storage unit**

character storage unit ([3.134.1](#)), *numeric storage unit* ([3.134.2](#)), *file storage unit* ([3.68](#)), or *unspecified storage unit* ([3.134.3](#))

Note 1 to entry: Storage units are described in [19.5.3.2](#).

3.134.1**character storage unit**

unit of storage that holds a default character value

3.134.2**numeric storage unit**

unit of storage that holds a default real, default integer, or default logical value

3.134.3**unspecified storage unit**

unit of storage that holds a value that is not default character, default real, double precision real, default integer, default logical, or default complex

3.135**stream file**

file composed of a sequence of file storage units

Note 1 to entry: See [12.1](#).

3.136**structure**

scalar *data object* ([3.42](#)) of *derived type* ([3.144.3](#))

3.136.1**structure component**

component ([3.30](#)) of a structure

3.136.2**structure constructor**

syntax that specifies a structure value or creates such a value

Note 1 to entry: The syntax of a structure constructor is defined by *structure-constructor* ([R756](#), [7.5.10](#)).

3.137**submodule**

program unit (3.113) that extends a *module* (3.99) or another submodule

Note 1 to entry: Submodules are described in 14.2.3.

3.138**subobject**

portion of *data object* (3.42) that can be referenced and, if it is a *variable* (3.151), defined, independently of any other portion

Note 1 to entry: The conditions for a structure component being a subobject are specified in 9.4.2.

3.139**subprogram**

function-subprogram (R1532) or *subroutine-subprogram* (R1537)

3.139.1**external subprogram**

subprogram that is not contained in a *main program* (3.97), *module* (3.99), *submodule* (3.137), or another subprogram

3.139.2**internal subprogram**

subprogram that is contained in a *main program* (3.97) or another subprogram

3.139.3**module subprogram**

subprogram that is contained in a *module* (3.99) or *submodule* (3.137) but is not an internal subprogram

3.140**subroutine**

procedure invoked by a *CALL statement*, by *defined assignment* (3.50), or by some operations on derived-type entities

3.140.1**atomic subroutine**

intrinsic subroutine that performs an action on its ATOM argument atomically

Note 1 to entry: Atomic subroutines are described in 16.5.

3.140.2**collective subroutine**

intrinsic subroutine that performs a calculation on a *team* (3.142) of images without requiring synchronization

Note 1 to entry: Collective subroutines are described in 16.6.

3.141**target**

entity that is [pointer associated](#) with a [pointer](#) (3.104), entity on the right-hand-side of a [pointer assignment statement](#), or entity with the [TARGET attribute](#)

Note 1 to entry: Pointer association is described in [19.5.2.2](#). The pointer assignment statement is described in [10.2.2](#). The TARGET attribute is described in [8.5.18](#).

3.142**team**

ordered set of [images](#) (3.80) created by execution of a [FORM TEAM statement](#), or the initial ordered set of all images

Note 1 to entry: The FORM TEAM statement is described in [11.7.9](#).

3.142.1**current team**

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

3.142.2**initial team**

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

3.142.3**parent team**

current team at time of execution of the [FORM TEAM statement](#) that created the team

Note 1 to entry: The initial team does not have a parent team.

3.142.4**sibling teams**

teams created by a single set of corresponding executions of the [FORM TEAM statement](#)

3.142.5**team number**

−1 which identifies the initial team, or positive integer which identifies a team among its siblings

3.143**transformational function**

intrinsic function, or function in an intrinsic module, that is neither [elemental](#) (3.62) nor an [inquiry function](#) (3.85)

3.144**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

Note 1 to entry: See [7.1](#).

3.144.1**abstract type**

type with the [ABSTRACT attribute](#)

Note 1 to entry: The ABSTRACT attribute is described in [7.5.7.1](#).

3.144.2**declared type**

type that a data entity is declared to have, either explicitly or implicitly

Note 1 to entry: See [7.3.2](#) and [10.1.9](#).

3.144.3**derived type**

type defined by a derived-type definition or by an intrinsic module

Note 1 to entry: See [7.5](#).

3.144.4**dynamic type**

type of a data entity at a particular point during execution of a program

Note 1 to entry: See [7.3.2.3](#) and [10.1.9](#).

3.144.5**extended type**

type with the [EXTENDS attribute](#)

Note 1 to entry: The EXTENDS attribute is described in [7.5.7.1](#).

3.144.6**extensible type**

type that can be extended using the EXTENDS clause

Note 1 to entry: See [7.5.7.1](#).

3.144.7**extension type**

is the same type or is an extended type whose parent type is an extension type of the other type

Note 1 to entry: This is a relation of one type with respect to another.

3.144.8**intrinsic type**

type defined by this document that is always accessible

Note 1 to entry: Intrinsic types are described in [7.4](#).

3.144.9**numeric type**

one of the types integer, real, and complex

3.144.10**parent type**

type named in the [EXTENDS](#) clause

Note 1 to entry: Only an extended type has a parent type.

3.144.11**type compatible**

compatibility of the type of one entity with respect to another for purposes such as [argument association](#) (3.7.1), [pointer association](#) (3.7.7), and allocation

Note 1 to entry: Type compatibility is described in 7.3.3.

3.144.12**type parameter**

value used to parameterize a type

Note 1 to entry: Type parameters are described in 7.2.

3.144.12.1**assumed type parameter**

[length type parameter](#) (3.144.12.4) that assumes the type parameter value from another entity

Note 1 to entry: The other entity is

- the selector for an [associate name](#) (3.5),
- the [constant-expr](#) for a [named constant](#) (3.34.2) of type character, or
- the [effective argument](#) (3.60) for a [dummy argument](#) (3.59).

3.144.12.2**deferred type parameter**

[length type parameter](#) (3.144.12.4) whose value can change during execution of a program and whose [type-param-value](#) is a colon

3.144.12.3**kind type parameter**

type parameter whose value is required to be defaulted or given by a [constant expression](#)

3.144.12.4**length type parameter**

type parameter whose value is permitted to be [assumed](#), [deferred](#), or given by a [specification expression](#) (3.128)

3.144.12.5**type parameter inquiry**

syntax ([type-param-inquiry](#)) that is used to inquire the value of a type parameter of a data object

Note 1 to entry: Type parameter enquiries are described in 9.4.5.

3.144.12.6**type parameter order**

ordering of the type parameters of a type used for derived-type specifiers

Note 1 to entry: Type parameter order is defined in 7.5.3.2. The syntax of a derived-type specifier is *derived-type-spec*, defined in 7.5.9.

3.145**ultimate argument**

nondummy entity with which a *dummy argument* (3.59) is associated via a chain of argument associations

Note 1 to entry: Argument association is described in 15.5.2.4.

3.146**undefined**

⟨data object⟩ without a valid value

3.147**undefined**

⟨pointer⟩ does not have a pointer association status of associated or disassociated

Note 1 to entry: Pointer association status is described in 19.5.2.2.

3.148**unit****input/output unit**

means, specified by an *io-unit*, for referring to a file

Note 1 to entry: See 12.5.1.

3.149**unlimited polymorphic**

able to have any *dynamic type* (3.144.4) during program execution

Note 1 to entry: See 7.3.2.3.

3.150**unsaved**

without the *SAVE attribute*

Note 1 to entry: The *SAVE attribute* is described in 8.5.16.

3.151**variable**

data entity (3.41) that can be *defined* (3.48) and redefined during execution of a program

3.151.1**event variable**

scalar variable of type *EVENT_TYPE* from the intrinsic module *ISO_FORTRAN_ENV*

Note 1 to entry: See 16.10.2.10.

3.151.2**local variable**

variable in a *scoping unit* (3.120) that is not a *dummy argument* (3.59) or part thereof, is not a global entity or part thereof, and is not an entity or part of an entity that is accessible outside that *scoping unit* (3.120)

3.151.3**lock variable**

scalar variable of type `LOCK_TYPE` from the intrinsic module `ISO_FORTRAN_ENV`

Note 1 to entry: See 16.10.2.19.

3.151.4**notify variable**

scalar variable of type `NOTIFY_TYPE` from the intrinsic module `ISO_FORTRAN_ENV`

Note 1 to entry: See 16.10.2.22.

3.151.5**team variable**

scalar variable of type `TEAM_TYPE` from the intrinsic module `ISO_FORTRAN_ENV`

Note 1 to entry: See 16.10.2.34.

3.152**vector subscript**

section-subscript that is an array

Note 1 to entry: See 9.5.3.4.3.

3.153**whole array**

array component or array name without further qualification

Note 1 to entry: See 9.5.2.

4 Notation, conformance, and compatibility

4.1 Notation, symbols and abbreviated terms

4.1.1 Syntax rules

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

- 3 Characters from the Fortran character set ([6.1](#)) are interpreted literally as shown, except where oth-
 4 erwise noted.
- 5 Lower-case italicized letters and words (often hyphenated and abbreviated) represent general syn-
 6 tactic classes for which particular syntactic entities shall be substituted in actual statements.
 7 Common abbreviations used in syntactic terms are:

<i>arg</i>	for	argument	<i>attr</i>	for	attribute
<i>decl</i>	for	declaration	<i>def</i>	for	definition
<i>desc</i>	for	descriptor	<i>expr</i>	for	expression
<i>int</i>	for	integer	<i>op</i>	for	operator
<i>spec</i>	for	specifier	<i>stmt</i>	for	statement

- 8 The syntactic metasympols 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

- 9 Each syntax rule is given a unique identifying number of the form Rsnn, where s is a one- or two-
 10 digit clause number and nn is a two-digit sequence number within that clause. The syntax rules are
 11 distributed as appropriate throughout the text, and are referenced by number as needed. Some rules
 12 in Clauses [5](#) and [6](#) are more fully described in later clauses; in such cases, the clause number s is the
 13 number of the later clause where the rule is repeated.
- 14 The syntax rules are not a complete and accurate syntax description of Fortran, and cannot be used
 15 to generate a Fortran parser automatically; where a syntax rule is incomplete, it is restricted by cor-
 16 responding constraints and text.

NOTE

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

Some examples of [digit-string](#) are:

NOTE (cont.)

4
67
1999
10243852

4.1.2 Constraints

- 1 Each constraint is given a unique identifying number of the form C_{snn}, where *s* is a one- or two-digit clause number and *nn* is a two- or three-digit sequence number within that clause.
- 2 Often a constraint is associated with a particular syntax rule. Where that is the case, the constraint is annotated with the syntax rule number in parentheses. A constraint that is associated with a syntax rule constitutes part of the definition of the syntax term defined by the rule. It thus applies in all places where the syntax term appears.
- 3 Some constraints are not associated with particular syntax rules. The effect of such a constraint is similar to that of a restriction stated in the text, except that a processor is required to have the capability to detect and report violations of constraints (4.2). In some cases, a broad requirement is stated in text and a subset of the same requirement is also stated as a constraint. This indicates that a standard-conforming program is required to adhere to the broad requirement, but that a standard-conforming processor is required only to have the capability of diagnosing violations of the constraint.

4.1.3 Assumed syntax rules

- 1 In order to minimize the number of additional syntax rules and convey appropriate constraint information, the following rules, where the letters *xyz* stand for any syntactic class phrase, are assumed.

R401	<i>xyz-list</i>	is	<i>xyz</i> [, <i>xyz</i>] ...
R402	<i>xyz-name</i>	is	<i>name</i>
R403	<i>scalar-xyz</i>	is	<i>xyz</i>
- 2 An explicit syntax rule for a term overrides an assumed rule.

4.1.4 Syntax conventions and characteristics

- 1 Any syntactic class name ending in “-*stmt*” follows the source form statement rules: it shall be delimited by end-of-line or semicolon, and may be labeled unless it forms part of another statement (such as an **IF** or **WHERE** statement). Conversely, everything considered to be a source form statement is given a “-*stmt*” ending in the syntax rules.
- 2 The rules on statement ordering are described rigorously in the definition of *program-unit* (R502). Expression hierarchy is described rigorously in the definition of *expr* (R1023).
- 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 R814), and in a few other cases.
- 4 Where reference is made to a type parameter, including the surrounding parentheses, the suffix “-*selector*” is used. See, for example, “*kind-selector*” (R706) and “*length-selector*” (R722).

4.1.5 Text conventions

- 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”.

4.2 Conformance

- 1 A **program** (5.2.2) is a standard-conforming program if it uses only those forms and relationships described herein and if the program has an interpretation according to this document. A **program unit** (5.2.1) conforms to this document if it can be included in a **program** in a manner that allows the **program** to be standard conforming.
- 2 A Fortran **processor** shall:
 - (1) execute 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) contain 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) contain the capability to detect and report the use within a submitted **program unit** of a form or relationship that is not permitted by the numbered syntax rules or constraints, including the deleted features described in Annex B;
 - (4) contain the capability to detect and report the use within a submitted **program unit** of an intrinsic type with a **kind type parameter** value not supported by the processor (7.4);
 - (5) contain the capability to detect and report the use within a submitted **program unit** of source form or characters not permitted by Clause 6;
 - (6) contain the capability to detect and report the use within a submitted **program** of name usage not consistent with the scope rules for names, labels, operators, and assignment symbols in Clause 19;
 - (7) contain the capability to detect and report the use within a submitted **program unit** of a non-standard intrinsic procedure (including one with the same name as a **standard intrinsic** procedure but with different requirements);
 - (8) contain the capability to detect and report the use within a submitted **program unit** of a non-standard intrinsic module;
 - (9) contain the capability to detect and report the use within a submitted **program unit** of a procedure from a **standard intrinsic** module, if the procedure is not defined by this document or the procedure has different requirements from those specified by this document; and
 - (10) contain 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** (13.2.1), a processor need not detect or report the use of deleted or obsolescent features, or the use of additional forms or relationships.
- 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 has the **EXTERNAL attribute** (8.5.9) where it is used. A standard-conforming program shall not use nonstandard intrinsic procedures or modules that have been added by the processor.

- 1
- 5
- Because a standard-conforming program may place demands on a processor that are not within the scope
- 2
- of this document or may include standard items that are not portable, such as [external procedures](#) defined
- 3
- by means other than Fortran, conformance to this document does not ensure that a program will execute
- 4
- consistently on all or any standard-conforming processors.
- 5
- 6
- The semantics of facilities that are identified as [processor dependent](#) are not completely specified in this
- 6
- document. They shall be provided, with methods or semantics determined by the processor.
- 7
- 7
- The [processor](#) should be accompanied by documentation that specifies the limits it imposes on the size
- 8
- and complexity of a [program](#) and the means of reporting when these limits are exceeded, that defines the
- 9
- additional forms and relationships it allows, and that defines the means of reporting the use of additional
- 10
- forms and relationships and the use of deleted or obsolescent forms. In this context, the use of a deleted
- 11
- form is the use of an additional form.
- 12
- 8
- The [processor](#) should be accompanied by documentation that specifies the methods or semantics of proces-
- 13
- sor-dependent facilities.

4.3 Compatibility

4.3.1 Previous Fortran standards

- 16
- 1
- Table [4.3](#) lists the previous editions of the Fortran International Standard, along with their informal names.

Table 4.3 — 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
ISO/IEC 1539-1:2018	Fortran 2018
ISO/IEC 1539-1:2023	Fortran 2023

4.3.2 New intrinsic procedures

- 18
- 1
- Each Fortran International Standard since ISO 1539:1980 (FORTRAN 77), defines more intrinsic procedures
- 19
- than the previous one. Therefore, a Fortran program conforming to an older standard might have a different
- 20
- interpretation under a newer standard if it invokes an external procedure having the same name as one of
- 21
- the new standard intrinsic procedures, unless that procedure is specified to have the [EXTERNAL attribute](#).

4.3.3 Fortran 2023 compatibility

- 23
- 1
- This document is an upward compatible extension to the preceding Fortran International Standard, ISO/IEC
- 24
- 1539-1:2023 (Fortran 2023). A standard-conforming Fortran 2023 program remains standard-conforming
- 25
- under this document.

4.3.4 Fortran 2018 compatibility

- 27
- 1
- Except as identified in this subclause, this document is an upward compatible extension to ISO/IEC 1539-
- 28
- 1:2018 (Fortran 2018). A standard-conforming Fortran 2018 program that does not use any feature iden-
- 29
- tified in this subclause as being no longer permitted remains standard-conforming under this document.

- 1 2 Fortran 2018 allowed integer arguments to the intrinsic subroutine `SYSTEM_CLOCK` to be of any kind. This
 2 document requires integer arguments to `SYSTEM_CLOCK` to have a decimal exponent range at least as large
 3 as a default integer, and requires that all integer arguments in a reference to `SYSTEM_CLOCK` have the same
 4 kind type parameter.
- 5 3 Fortran 2018 permitted a variable in a `BLOCK construct` that was declared only by a `DATA statement` to be
 6 used before the `DATA statement`. This document does not permit such usage.
- 7 4 Fortran 2018 permitted the `POINTER` and `TARGET` arguments to the intrinsic function `ASSOCIATED` to have
 8 different rank; this document does not permit such usage.
- 9 5 Fortran 2018 permitted a *component-data-source* in a structure constructor to be a `coindexed object` with
 10 a pointer subcomponent. This document does not permit such usage.
- 11 6 The following Fortran 2018 features might have a different interpretation under this document.
- 12 • After an allocatable deferred length character variable is assigned a value by an `IOMSG=` or `ERRMSG=`
 13 clause, is the unit in an internal `WRITE statement`, or is an `INTENT (OUT)` argument in a reference to
 14 an intrinsic subroutine, that variable might be of shorter or longer length under this document than
 15 under Fortran 2018, since this document specifies intrinsic assignment semantics for these assign-
 16 ments.
 - 17 • This document permits the intrinsic subroutine `SYSTEM_CLOCK` to use two or more clocks, with dif-
 18 ferent characteristics based on the type and kind type parameters of its arguments. A program that
 19 invokes `SYSTEM_CLOCK` with different argument types or kinds in different references, could have a
 20 different interpretation under this document.
 - 21 • The result of a reference to `IEEE_MAX_NUM`, `IEEE_MAX_NUM_MAG`, `IEEE_MIN_NUM`, or `IEEE_MIN_-`
 22 `NUM_MAG` where one argument is a number and the other is a signaling NaN is specified to be the
 23 number in this document. Fortran 2018 specified that the result is a NaN.
 - 24 • The result of a reference to the function `IEEE_SCALB` (X, I) from the intrinsic module `IEEE_ARITH-`
 25 `METIC` has been corrected to return the representable number having a magnitude nearest to $|X \times 2^I|$
 26 with the same sign as X, if $X \times 2^I$ is too small to be represented with full accuracy.

27 4.3.5 Fortran 2008 compatibility

- 28 1 Except as identified in this subclause, and except for the deleted features noted in Clause B.2, this document
 29 is an upward compatible extension to ISO/IEC 1539-1:2010 (Fortran 2008). Any standard-conforming
 30 Fortran 2008 program that does not use any deleted features, and does not use any feature identified in
 31 this subclause as being no longer permitted, remains standard-conforming under this document.
- 32 2 Fortran 2008 specifies that the `IOSTAT=` variable shall be set to a processor-dependent negative value if
 33 the flush operation is not supported for the unit specified. This document specifies that the processor-
 34 dependent negative integer value shall be different from the `named constants` `IOSTAT_EOR` or `IOSTAT_END`
 35 from the intrinsic module `ISO_FORTRAN_ENV`.
- 36 3 Fortran 2008 permitted a noncontiguous array that was supplied as an `actual argument` corresponding to
 37 a `contiguous INTENT (INOUT) dummy argument` in one iteration of a `DO CONCURRENT construct`, without
 38 being previously defined in that iteration, to be defined in another iteration; this document does not permit
 39 this.
- 40 4 Fortran 2008 permitted a `pure` statement function to reference a `volatile` variable, and permitted a `local`
 41 `variable` of a `pure` subprogram or of a `BLOCK construct` within a `pure` subprogram to be `volatile` (provided
 42 it was not used); this document does not permit that.

- 1 5 Fortran 2008 permitted a **pure** function to have a result that has a **polymorphic** allocatable **ultimate com-**
 2 **ponent**; this document does not permit that.
- 3 6 Fortran 2008 permitted a **PROTECTED TARGET** variable accessed by **use association** to be used as an **initial-**
 4 **data-target**; this document does not permit that.
- 5 7 Fortran 2008 permitted a **named constant** to have declared type **LOCK_TYPE**, or have a noncoarray **poten-**
 6 **tial subobject component** with declared type **LOCK_TYPE**; this document does not permit that.
- 7 8 Fortran 2008 permitted a **polymorphic** object to be finalized within a **DO CONCURRENT construct**; this
 8 document does not permit that.
- 9 9 Fortran 2008 permitted an unallocated **allocatable coarray** or **coindexed object** to be allocated by an **as-**
 10 **signment statement**, provided it was scalar, nonpolymorphic, and had no **deferred type parameters**; this
 11 document does not permit that.
- 12 10 Fortran 2008 permitted the processor to use a common pseudorandom number generator for all **images**.
 13 This document requires separate seeds on each **image** for the pseudorandom number generator.
- 14 11 Fortran 2008 required **ACOSH** of a complex value to have the imaginary part nonnegative and had no re-
 15 quirement on the real part. This document requires **ACOSH** of a complex value to have a nonnegative real
 16 part and has no such requirement on the imaginary part.
- 17 12 Fortran 2008 allowed integer arguments to the intrinsic subroutine **SYSTEM_CLOCK** to be of any kind. This
 18 document requires integer arguments to **SYSTEM_CLOCK** to have a decimal exponent range at least as large
 19 as a default integer, and requires that all integer arguments in a reference to **SYSTEM_CLOCK** have the same
 20 kind type parameter.
- 21 13 Fortran 2008 permitted a variable in a **BLOCK construct** that was declared only by a **DATA statement** to be
 22 used before the **DATA statement**. This document does not permit such usage.
- 23 14 Fortran 2008 permitted the **POINTER** and **TARGET** arguments to the intrinsic function **ASSOCIATED** to have
 24 different rank; this document does not permit such usage.
- 25 15 Fortran 2008 permitted a **component-data-source** in a structure constructor to be a **coindexed object** with
 26 a pointer subcomponent. This document does not permit such usage.
- 27 16 The following Fortran 2008 features might have a different interpretation under this document.
- 28 • After an allocatable deferred length character variable is assigned a value by an **IOMSG=** or **ERRMSG=**
 29 clause, is the unit in an internal **WRITE statement**, or is an **INTENT (OUT)** argument in a reference to
 30 an intrinsic subroutine, that variable might be of shorter or longer length under this document than
 31 under Fortran 2008, since this document specifies intrinsic assignment semantics for these assign-
 32 ments.
 - 33 • This document permits the intrinsic subroutine **SYSTEM_CLOCK** to use two or more clocks, with dif-
 34 ferent characteristics based on the type and kind type parameters of its arguments. A program that
 35 invokes **SYSTEM_CLOCK** with different argument types or kinds in different references, could have a
 36 different interpretation under this document.
 - 37 • The result of a reference to the function **IEEE_SCALB** (X, I) from the intrinsic module **IEEE_ARITH-**
 38 **METIC** has been corrected to return the representable number having a magnitude nearest to $|X \times 2^I|$
 39 with the same sign as X, if $X \times 2^I$ is too small to be represented with full accuracy.

4.3.6 Fortran 2003 compatibility

1 Except as identified in this subclause, this document is an upward compatible extension to ISO/IEC 1539-1:2004 (Fortran 2003). Except as identified in this subclause, any standard-conforming Fortran 2003 program remains standard-conforming under this document.

2 Fortran 2003 permitted a [sequence type](#) to have [type parameters](#); that is not permitted by this document.

3 Fortran 2003 specified that array constructors and structure constructors of [finalizable](#) type are finalized. This document specifies that these constructors are not finalized.

4 The form produced by the G edit descriptor for some values and some input/output rounding modes differs from that specified by Fortran 2003.

5 Fortran 2003 required an [explicit interface](#) only for a procedure that was actually referenced in the scope, not merely passed as an [actual argument](#). This document requires an [explicit interface](#) for a procedure under the conditions listed in [15.4.2.2](#), 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, to have a [polymorphic](#) allocatable [ultimate component](#), or to be [finalizable](#) by an impure [final subroutine](#). These are not permitted by this document.

7 Fortran 2003 permitted an [INTENT \(OUT\)](#) argument of a [pure](#) subroutine to be [polymorphic](#); that is not permitted by this document.

8 Fortran 2003 interpreted assignment to an allocatable variable from a nonconformable array as [intrinsic assignment](#), even when an elemental [defined assignment](#) was in scope; this document does not permit assignment from a nonconformable array in this context.

9 Fortran 2003 permitted a statement function to be of parameterized derived type; this document does not permit that.

10 Fortran 2003 permitted a [pure](#) statement function to reference a [volatile](#) variable, and permitted a [local variable](#) of a [pure](#) subprogram to be [volatile](#) (provided it was not used); this document does not permit that.

11 Fortran 2003 allowed integer arguments to the intrinsic subroutine [SYSTEM_CLOCK](#) to be of any kind. This document requires integer arguments to [SYSTEM_CLOCK](#) to have a decimal exponent range at least as large as a default integer, and requires that all integer arguments in a reference to [SYSTEM_CLOCK](#) have the same kind type parameter.

12 Fortran 2003 permitted the POINTER and TARGET arguments to the intrinsic function [ASSOCIATED](#) to have different rank; this document does not permit such usage.

13 The following Fortran 2003 features might have a different interpretation under this document.

- After an allocatable deferred length character variable is assigned a value by an [IOMSG=](#) or [ERRMSG=](#) clause, is the unit in an internal [WRITE statement](#), or is an [INTENT \(OUT\)](#) argument in a reference to an intrinsic subroutine, that variable might be of shorter or longer length under this document than under Fortran 2003, since this document specifies intrinsic assignment semantics for these assignments.
- This document permits the intrinsic subroutine [SYSTEM_CLOCK](#) to use two or more clocks, with different characteristics based on the type and kind type parameters of its arguments. A program that invokes [SYSTEM_CLOCK](#) with different argument types or kinds in different references, could have a

different interpretation under this document.

- The result of a reference to the function `IEEE_SCALB` (X, I) from the intrinsic module `IEEE_ARITHMETIC` has been corrected to return the representable number having a magnitude nearest to $|X \times 2^I|$ with the same sign as X, if $X \times 2^I$ is too small to be represented with full accuracy.

4.3.7 Fortran 95 compatibility

1 Except as identified in this subclause, this document is an upward compatible extension to ISO/IEC 1539-1:1997 (Fortran 95). Except as identified in this subclause, any standard-conforming Fortran 95 program remains standard-conforming under this document.

2 Fortran 95 permitted defined assignment between character strings of the same rank and different kinds. This document does not permit that if both of the different kinds are ASCII, ISO 10646, or default kind.

3 The following Fortran 95 features might have different interpretations in this document.

- Earlier Fortran standards had the concept of printing, meaning that column one of formatted output had special meaning for a processor-dependent (possibly empty) set of `external files`. This could be neither detected nor specified by a standard-specified means. The interpretation of the first column is not specified by this document.
- This document specifies a different output format for real zero values in list-directed and namelist output.
- If the processor distinguishes between positive and negative real zero, this document requires different returned values for `ATAN2(Y,X)` when $X < 0$ and Y is negative real zero and for `LOG(X)` and `SQRT(X)` when X is complex with $X\%RE < 0$ and $X\%IM$ is negative real zero.
- This document has fewer restrictions on `constant expressions` than Fortran 95; this affects whether a variable is considered to be an `automatic data object`.
- The form produced by the G edit descriptor with *d* equal to zero differs from that specified by Fortran 95 for some values.

4.3.8 Fortran 90 compatibility

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

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

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

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

5 The following Fortran 90 features have different interpretations in this document:

- if the processor distinguishes between positive and negative real zero, the result value of the intrinsic function `SIGN` when the second argument is a negative real zero;
- formatted output of negative real values (when the output value is zero);

- whether an expression is a [constant expression](#) (thus whether a variable is considered to be an [automatic data object](#));
- the G edit descriptor with *d* equal to zero for some values.

4.3.9 FORTRAN 77 compatibility

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

- FORTRAN 77 permitted a processor to supply more precision derived from a default real constant than can be represented in a default real datum when the constant is used to initialize a double precision real data object in a [DATA statement](#). This document does not permit a processor this option.
- If a named variable that was not in a [common block](#)¹ was initialized in a [DATA statement](#) and did not have the [SAVE attribute](#) specified, FORTRAN 77 left its [SAVE attribute](#) processor dependent. This document specifies ([8.6.7](#)) that this named variable has the [SAVE attribute](#).
- FORTRAN 77 specified that the number of characters required by the input list was to be less than or equal to the number of characters in the record during formatted input. This document specifies ([12.6.4.5.3](#)) that the input record is logically padded with blanks if there are not enough characters in the record, unless the [PAD= specifier](#) with the value 'NO' is specified in an appropriate [OPEN](#) or [READ](#) statement.
- A value of zero for an [effective item](#) in a formatted [output statement](#) will be formatted in a different form for some G edit descriptors. In addition, this document specifies how rounding of values will affect the output field form, but FORTRAN 77 did not address this issue. Therefore, the form produced for certain combinations of values and G edit descriptors might differ from that produced by some FORTRAN 77 processors.
- FORTRAN 77 did not permit a processor to distinguish between positive and negative real zero; if the processor does so distinguish, the result will differ for the intrinsic function [SIGN](#) when the second argument is negative real zero, and formatted output of negative real zero will be different.

4.4 Deleted and obsolescent features

4.4.1 General

1 This document protects the users' investment in existing software by including all but eight of the language elements of Fortran 90 that are not processor dependent. This document identifies two categories of outmoded features. The first category, deleted features, consists of features considered to have been redundant in Fortran 90 and largely unused. Those in the second category, obsolescent features, are considered to have been redundant in Fortran 2008, but are still frequently used.

¹Common blocks are obsolescent.

1 **4.4.2 Nature of deleted features**

2 1 There are two groups of deleted features. The first group contains features for which better methods exis-
3 ted in FORTRAN 77; these features were not included in Fortran 95 and later revisions, and are not included
4 in this document. The second group contains features for which better methods existed in Fortran 90; these
5 features were not included in Fortran 2018 and later revisions, and are not included in this document.

6 **4.4.3 Nature of obsolescent features**

7 1 Better methods existed in a previous Fortran standard for each obsolescent feature. It is recommended
8 that programmers use these better methods in new programs and convert existing code to these methods.
9 2 An obsolescent feature is eligible for deletion from a future revision of this document if its use has become
10 insignificant.

5 Fortran concepts

5.1 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

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

R501	<i>program</i>	is	<i>program-unit</i> [<i>program-unit</i>] ...
R502	<i>program-unit</i>	is	<i>main-program</i> or <i>external-subprogram</i> or <i>module</i> or <i>submodule</i> or <i>block-data</i>
R1401	<i>main-program</i>	is	[<i>program-stmt</i>] [<i>specification-part</i>] [<i>execution-part</i>] [<i>internal-subprogram-part</i>] <i>end-program-stmt</i>
R503	<i>external-subprogram</i>	is	<i>function-subprogram</i> or <i>subroutine-subprogram</i>
R1532	<i>function-subprogram</i>	is	<i>function-stmt</i> [<i>specification-part</i>] [<i>execution-part</i>] [<i>internal-subprogram-part</i>] <i>end-function-stmt</i>
R1537	<i>subroutine-subprogram</i>	is	<i>subroutine-stmt</i> [<i>specification-part</i>] [<i>execution-part</i>] [<i>internal-subprogram-part</i>] <i>end-subroutine-stmt</i>
R1404	<i>module</i>	is	<i>module-stmt</i> [<i>specification-part</i>] [<i>module-subprogram-part</i>] <i>end-module-stmt</i>
R1416	<i>submodule</i>	is	<i>submodule-stmt</i> [<i>specification-part</i>] [<i>module-subprogram-part</i>]

1			<i>end-submodule-stmt</i>
2	R1420	<i>block-data</i>	is <i>block-data-stmt</i>
3			[<i>specification-part</i>]
4			<i>end-block-data-stmt</i>
5	R504	<i>specification-part</i>	is [<i>use-stmt</i>] ...
6			[<i>import-stmt</i>] ...
7			[<i>implicit-part</i>]
8			[<i>declaration-construct</i>] ...
9	R505	<i>implicit-part</i>	is [<i>implicit-part-stmt</i>] ...
10			<i>implicit-stmt</i>
11	R506	<i>implicit-part-stmt</i>	is <i>implicit-stmt</i>
12			or <i>parameter-stmt</i>
13			or <i>format-stmt</i>
14			or <i>entry-stmt</i>
15	R507	<i>declaration-construct</i>	is <i>specification-construct</i>
16			or <i>data-stmt</i>
17			or <i>format-stmt</i>
18			or <i>entry-stmt</i>
19			or <i>stmt-function-stmt</i>
20	R508	<i>specification-construct</i>	is <i>derived-type-def</i>
21			or <i>enum-def</i>
22			or <i>enumeration-type-def</i>
23			or <i>generic-stmt</i>
24			or <i>interface-block</i>
25			or <i>parameter-stmt</i>
26			or <i>procedure-declaration-stmt</i>
27			or <i>other-specification-stmt</i>
28			or <i>type-declaration-stmt</i>
29	R509	<i>execution-part</i>	is <i>executable-construct</i>
30			[<i>execution-part-construct</i>] ...
31	R510	<i>execution-part-construct</i>	is <i>executable-construct</i>
32			or <i>format-stmt</i>
33			or <i>entry-stmt</i>
34			or <i>data-stmt</i>
35	R511	<i>internal-subprogram-part</i>	is <i>contains-stmt</i>
36			[<i>internal-subprogram</i>] ...
37	R512	<i>internal-subprogram</i>	is <i>function-subprogram</i>
38			or <i>subroutine-subprogram</i>
39	R1407	<i>module-subprogram-part</i>	is <i>contains-stmt</i>
40			[<i>module-subprogram</i>] ...


```

1  R1408 module-subprogram      is function-subprogram
2                                or subroutine-subprogram
3                                or separate-module-subprogram
4
5  R1541 separate-module-subprogram is mp-subprogram-stmt
6                                [ specification-part ]
7                                [ execution-part ]
8                                [ internal-subprogram-part ]
9                                end-mp-subprogram-stmt
10
11 R513  other-specification-stmt is access-stmt
12                                or allocatable-stmt
13                                or asynchronous-stmt
14                                or bind-stmt
15                                or codimension-stmt
16                                or contiguous-stmt
17                                or dimension-stmt
18                                or external-stmt
19                                or intent-stmt
20                                or intrinsic-stmt
21                                or namelist-stmt
22                                or optional-stmt
23                                or pointer-stmt
24                                or protected-stmt
25                                or save-stmt
26                                or target-stmt
27                                or volatile-stmt
28                                or value-stmt
29                                or common-stmt
30                                or equivalence-stmt
31
32 R514  executable-construct    is action-stmt
33                                or associate-construct
34                                or block-construct
35                                or case-construct
36                                or change-team-construct
37                                or critical-construct
38                                or do-construct
39                                or if-construct
40                                or select-rank-construct
41                                or select-type-construct
42                                or where-construct
43                                or forall-construct
44
45 R515  action-stmt             is allocate-stmt
46                                or assignment-stmt
47                                or backspace-stmt
48                                or call-stmt
49                                or close-stmt
50                                or continue-stmt

```

or *cycle-stmt*
 or *deallocate-stmt*
 or *endfile-stmt*
 or *error-stop-stmt*
 or *event-post-stmt*
 or *event-wait-stmt*
 or *exit-stmt*
 or *fail-image-stmt*
 or *flush-stmt*
 or *form-team-stmt*
 or *goto-stmt*
 or *if-stmt*
 or *inquire-stmt*
 or *lock-stmt*
 or *notify-wait-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 *sync-team-stmt*
 or *unlock-stmt*
 or *wait-stmt*
 or *where-stmt*
 or *write-stmt*
 or *computed-goto-stmt*
 or *forall-stmt*

5.2 Program unit concepts

5.2.1 Program units and scoping units

- 1 **Program units** are the fundamental components of a Fortran program. A **program unit** is a **main program**,
 an **external subprogram**, a **module**, a **submodule**, or a block data program unit.
- 2 A **subprogram** is a function subprogram or a subroutine subprogram. A **module** contains definitions that
 can be made accessible to other **program units**. A **submodule** is an extension of a **module**; it may contain
 the definitions of procedures declared in a **module** or another **submodule**. A block data program unit is
 used to specify initial values for **data objects** in named **common blocks**. The block data program unit is
 obsolescent.
- 3 Each type of **program unit** is described in Clause 14 or 15.
- 4 A **program unit** consists of a set of nonoverlapping **scoping units**.

NOTE

The module or submodule containing a [module subprogram](#) is the [host scoping unit](#) of the [module subprogram](#). The containing [main program](#) or [subprogram](#) is the [host scoping unit](#) of an [internal subprogram](#).

An [internal procedure](#) is local to its [host](#) in the sense that its name is accessible within the [host scoping unit](#) and all its other [internal procedures](#) but is not accessible elsewhere.

5.2.2 Program

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

5.2.3 Procedure

- 1 A procedure is either a function or a subroutine. Invocation of a function in an expression causes a value to be computed which is then used in evaluating the expression.
- 2 A procedure that is not pure may change the program state by changing the value of accessible [data objects](#) or [procedure pointers](#).
- 3 Procedures are described further in Clause [15](#).

5.2.4 Module

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

5.2.5 Submodule

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

NOTE

A submodule has access to entities in its parent module or submodule by [host association](#).

5.3 Execution concepts**5.3.1 Statement classification**

- 1 Each Fortran statement is classified as either an [executable statement](#) or a [nonexecutable statement](#).
- 2 An [executable statement](#) is an instruction to perform or control an action. Thus, the executable statements of a [program unit](#) determine the behavior of the [program unit](#).
- 3 [Nonexecutable statements](#) are used to configure the program environment in which actions take place.

5.3.2 Statement order

Table 5.1 — Requirements on statement ordering

PROGRAM, FUNCTION, SUBROUTINE, MODULE, SUBMODULE, or BLOCK DATA statement		
USE statements		
IMPORT statements		
FORMAT and ENTRY statements	IMPLICIT NONE	
	PARAMETER statements	IMPLICIT statements
	PARAMETER and DATA statements	Specification constructs and statement function statements
	DATA statements	Executable constructs
CONTAINS statement		
Internal subprograms or module subprograms		
END statement		
NOTE 1 The BLOCK DATA, ENTRY, and statement function statements are obsolescent.		
NOTE 2 The appearance of a DATA statement after an executable statement is obsolescent.		

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

Table 5.2 — Statements allowed in scoping units

Statement type	Kind of scoping unit						
	Main program	Module or submodule	Block data ²	External subprogram	Module subprogram	Internal subprogram	Interface body
USE	Yes	Yes	Yes	Yes	Yes	Yes	Yes
IMPORT	No	Submodule	No	No	Yes	Yes	Yes
ENTRY ²	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 ²	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) The block data scoping unit, ENTRY statement , and statement function statements are obsolescent.							

5.3.3 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](#) (11.2). Executing an [end-program-stmt](#) initiates [normal termination](#). Executing an [end-function-stmt](#), [end-subroutine-stmt](#), or [end-mp-subprogram-stmt](#) is equivalent to executing a [return-stmt](#) with no [scalar-int-expr](#).
- 2 The [end-module-stmt](#), [end-submodule-stmt](#), and [end-block-data-stmt](#) statements are nonexecutable.

5.3.4 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 (17.7), and set of [data objects](#), [input/output units](#), and [procedure pointers](#). The [image index](#) that identifies an [image](#) is an integer value in the range one to the number of [images](#) in a [team](#).
- 2 A [team](#) is an ordered set of [images](#) that is either the [initial team](#), consisting of all [images](#), or a subset of a [parent team](#) formed by execution of a [FORM TEAM statement](#). The [initial team](#) has no [parent](#); every other team has a unique [parent team](#). Among its [sibling teams](#), each [team](#) is identified by its [team number](#); this is the integer value that was specified in the [FORM TEAM statement](#).
- 3 During execution, each image has a [current team](#), which is only changed by execution of [CHANGE TEAM](#) and [END TEAM](#) statements. Image indices, and thus coindexing of variable names with an [image-selector](#), are relative to the [current team](#) unless a different team is specified. Initially, the [current team](#) is the [initial team](#).

NOTE 1

Fortran control constructs (11.1, 11.2) control the progress of execution in each [image](#). [Image control statements](#) (11.7.1) affect the relative progress of execution between [images](#). [Coarrays](#) (5.4.7) provide a mechanism for accessing data on one [image](#) from another [image](#).

NOTE 2

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

5.3.5 Execution sequence

- 1 Following the creation of a fixed number of [images](#), execution begins on each [image](#). [Image](#) execution is a sequence, in time, of actions. Actions take place during execution of the statement that performs them (except when explicitly stated otherwise). Segments (11.7.2) executed by a single [image](#) are totally ordered, and segments executed by separate [images](#) are partially ordered by [image control statements](#) (11.7.1).
- 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 (**11.2**) changes the execution sequence. These statements explicitly specify a new starting place for the execution sequence.
- **DO constructs**, **IF constructs**, **SELECT CASE constructs**, **SELECT RANK constructs**, and **SELECT TYPE constructs** contain an internal statement structure and execution of these constructs involves implicit internal transfer of control. See Clause **11** for the detailed semantics of each of these constructs.
- A **BLOCK construct** may contain **specification expressions**; see **11.1.4** for detailed semantics of this construct.
- An **END=**, **ERR=**, or **EOR=** specifier (**12.11**) can result in a branch.
- An alternate return can result in a branch.¹

5.3.6 Image execution states

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

2 A **failed image** remains failed for the remainder of the execution of the program. The conditions that cause an **image** to fail are processor dependent. It is processor dependent whether the processor has the ability to detect that an image has failed.

3 Defining a **coindexed object** on a **failed image** has no effect other than defining the **stat-variable**, if one appears, with the value **STAT_FAILED_IMAGE** (**16.10.2.28**). The value of a reference to a **coindexed object** on a **failed image** is processor dependent. Execution continues after such a reference.

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

5.3.7 Termination of execution

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 or failed.

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** can also be initiated during execution of a procedure defined by a **companion processor** (ISO/IEC 9899:2018, 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 9899:2018, 7.22.4.4) during the completion step.

¹Alternate returns are obsolescent.

- 1 3 Error termination of execution of an **image** is initiated if an **ERROR STOP statement** is executed or as specified elsewhere in this document. When error termination on an **image** has been initiated, the processor should initiate error termination on other **images** as quickly as possible.
- 4 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 1

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

NOTE 2

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

7 **5.4 Data concepts**

8 **5.4.1 Type**

9 **5.4.1.1 General**

- 10 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 7.1.
- 13 2 A **type** is either an **intrinsic type** or a nonintrinsic type. A nonintrinsic type is defined by the program or by an **intrinsic** module.

15 **5.4.1.2 Intrinsic type**

- 16 1 The **intrinsic types** are integer, real, complex, character, and logical. The properties of **intrinsic types** are described in 7.4.
- 18 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.

21 **5.4.1.3 Derived type**

- 22 1 **Derived types** can be parameterized. A scalar **object** of **derived type** is a **structure**; assignment of structures is defined intrinsically (10.2.1.3), but there are no **intrinsic** operations for structures. For each **derived type**, a **structure constructor** is available to create values (7.5.10). In addition, **objects** of **derived type** can be used as procedure arguments and function results, and can appear in input/output lists. If additional operations are needed for a **derived type**, they can be defined by procedures (10.1.6).
- 27 2 **Derived types** are described further in 7.5.

28 **5.4.2 Data value**

- 29 1 Each **intrinsic type** has associated with it a set of values that a datum of that type can take, depending on the values of the type parameters. The values for each **intrinsic type** are described in 7.4. The values that

objects of a **derived type** can assume are determined by the type definition, type parameter values, and the sets of values of its components. The values that an object of a nonderived nonintrinsic type can assume are determined by the type definition.

5.4.3 Data entity

5.4.3.1 General

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

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

5.4.3.2 Data object

5.4.3.2.1 Data object classification

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** can be specified explicitly (8.2) or implicitly (8.7).

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

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

4 The following **objects** are referenced by a name:

- a named scalar (a scalar object);
- a named array (an array object).

5 The following **subobjects** are referenced by an **object designator**:

- an array element (a scalar subobject);
- an array section (an array subobject);
- a **complex part designator** (the real or imaginary part of a complex object);
- a structure component (a scalar or an array subobject);
- a substring (a scalar subobject).

5.4.3.2.2 Variable

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

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

NOTE

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. Note that **common blocks** are obsolescent.

5.4.3.2.3 Constant

1 A constant is either a [named constant](#) or a [literal constant](#).

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

5.4.3.2.4 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

For example, given:

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

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

5.4.3.3 Expression

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

5.4.3.4 Function reference

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

5.4.4 Definition of objects and pointers

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

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

3 Clause 19 describes the ways in which variables become [defined](#) and [undefined](#) and the association status of pointers becomes [defined](#) and [undefined](#).

5.4.5 Reference

1 A [data object](#) is [referenced](#) when its value is required during execution. A procedure is [referenced](#) when it is executed.

1 2 The appearance of a [data object designator](#) or [procedure designator](#) as an [actual argument](#) does not con-
 2 stitute a [reference](#) to that [data object](#) or procedure unless such a [reference](#) is necessary to complete the
 3 specification of the [actual argument](#).

4 **5.4.6 Array**

5 1 An array may have up to fifteen dimensions minus its [corank](#), and any [extent](#) in any dimension. The [size](#)
 6 of an array is the total number of elements, which is equal to the product of the [extents](#). An array may
 7 have zero size. The [shape](#) of an array is determined by its [rank](#) and its [extent](#) in each dimension, and is
 8 represented as a rank-one array whose elements are the extents. All named arrays shall be declared, and
 9 the [rank](#) of a named array is specified in its declaration. Except for an [assumed-rank](#) array, the [rank](#) of a
 10 named array, once declared, is constant.

11 2 Any [intrinsic](#) operation defined for scalar [objects](#) may be applied to [conformable objects](#). Such operations
 12 are performed [elementally](#) to produce a resultant array [conformable](#) with the array operands. If an [ele-](#)
 13 [mental](#) operation is intrinsically pure or is implemented by a pure [elemental function](#) (15.9), the element
 14 operations can be performed simultaneously or in any order.

15 3 A rank-one array can be constructed from scalars and other arrays and can be reshaped into any allowable
 16 array shape (7.8).

17 4 Arrays are described further in 9.5.

18 **5.4.7 Coarray**

19 1 A [coarray](#) is a [component](#) (7.5.4.3), or [variable](#) (9.2), that has nonzero [corank](#). A coarray variable can be
 20 directly referenced or defined by other [images](#). It may be a scalar or an array.

21 2 Requirements and semantics for [coarrays](#) that refer to properties that are possessed by [variables](#), but not
 22 by type [components](#), only apply to [coarray variables](#).

23 3 For each [established coarray](#) on an [image](#), there is a corresponding [coarray](#) with the same type, type para-
 24 meters, and [bounds](#) on every other [image](#) of a [team](#) in which it is established (5.4.8). For each unallocated
 25 coarray, there exists a corresponding unallocated coarray with the same declared type, rank, corank, and
 26 non-deferred type parameters on each [active image](#) of the [current team](#).

27 4 For a named [coarray](#) that is not a [dummy argument](#), its corresponding coarrays are the ones with the same
 28 name in that [scoping unit](#). For a [coarray](#) that is a [component](#) at any level of component selection, each
 29 corresponding coarray is the same [component](#) of the [base object](#) that has the same name in that [scoping](#)
 30 [unit](#). If a [coarray component](#) is a [potential subobject component](#) of an [array element](#), the [array element](#) for
 31 each corresponding coarray has the same position in [array element order](#) on each [image](#).

32 5 If a [coarray](#) is an unsaved [local variable](#) of a recursive procedure, its corresponding [coarrays](#) are the ones
 33 at the same depth of recursion of that procedure on each [image](#).

34 6 The set of corresponding [established coarrays](#) on all [images](#) in a [team](#) is arranged in a rectangular pat-
 35 tern. The dimensions of this pattern are the [codimensions](#); the number of [codimensions](#) is the [corank](#). The
 36 bounds for each [codimension](#) are the [cobounds](#).

NOTE 1

If the total number of [images](#) is not a multiple of the product of the sizes of each but the rightmost of the [codimen-](#)
[sions](#), the rectangular pattern will be incomplete.

- 1 7 A **coarray** on any **image** can be accessed directly by using **cosubscripts**. On its own **image**, a **coarray** can
2 also be accessed without use of **cosubscripts**.
- 3 8 A **subobject** of a **coarray** is a **coarray** if it does not have any **cosubscripts**, **vector subscripts**, **allocatable**
4 **component selection**, or **pointer component selection**.
- 5 9 For a **coindexed object**, its **cosubscript** list determines the **image index** (9.6) in the same way that a subscript
6 list determines the subscript order value for an **array element** (9.5.3.3).
- 7 10 **Intrinsic** procedures are provided for mapping between an **image index** and a list of **cosubscripts**.

NOTE 2

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 3

Except in contexts where **coindexed objects** are disallowed, accessing a **coarray** on its own **image** by using a set of **cosubscripts** that specify that **image** has the same effect as accessing it without **cosubscripts**. In particular, the segment ordering rules (11.7.2) apply whether or not **cosubscripts** are used to access the **coarray**.

8 5.4.8 Established coarrays

- 9 1 A nonallocatable **coarray** with the **SAVE attribute** is **established** in the **initial team**.
- 10 2 An allocated allocatable **coarray** is **established** in the **team** in which it was allocated. An unallocated alloc-
11 atable **coarray** is not established.
- 12 3 A **coarray** that is **established** in the team in which a **CHANGE TEAM statement** is executed is **established** in
13 the team of the **CHANGE TEAM construct**.
- 14 4 A **coarray** that is an **associating entity** in a **coarray-association** of a **CHANGE TEAM statement** is **established**
15 in the **team** of its **CHANGE TEAM construct**.
- 16 5 A nonallocatable **coarray** that is an **associating entity** in an **ASSOCIATE**, **SELECT RANK**, or **SELECT TYPE**
17 **construct** is **established** in the team in which the **ASSOCIATE**, **SELECT RANK**, or **SELECT TYPE statement** is
18 executed.
- 19 6 A nonallocatable **coarray** that is a **dummy argument** or host associated with a **dummy argument** is **estab-**
20 **lished** in the **team** in which the procedure was invoked. A nonallocatable **coarray dummy argument** is not
21 **established** in any **ancestor team** even if the corresponding **actual argument** is **established** in one or more
22 of them.

23 5.4.9 Pointer

- 24 1 A **pointer** has an association status which is either associated, **disassociated**, or undefined (19.5.2.2).
- 25 2 A **pointer** that is not associated shall not be referenced or defined.
- 26 3 If a **data pointer** is an **array**, the **rank** is declared, but the **bounds** are determined when it is associated with
27 a **target**.

5.4.10 Allocatable variables

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

5.4.11 Storage

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

5.5 Fundamental concepts

5.5.1 Names and designators

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

5.5.2 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.

5.5.3 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 can appear in an **actual argument** list. A **type parameter keyword** is the name of a type parameter in the type being specified, and can appear in a **type-param-spec**. A **component keyword** is the name of a component in a **structure constructor**.

R516 *keyword* **is** *name*

NOTE

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

5.5.4 Association

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

- 1 2 [Pointer association](#) (19.5.2) between a pointer and a target allows the target to be denoted by the pointer.
- 2 3 [Storage association](#) (19.5.3) causes different entities to use the same storage.
- 3 4 [Inheritance association](#) (19.5.4) occurs between components of the parent component and components
4 inherited by type extension.

5 5.5.5 Intrinsic

- 6 1 All [intrinsic](#) types, procedures, assignments, and operators may be used in any [scoping unit](#) without further
7 definition or specification. [Intrinsic](#) modules (16.10, 17, 18.2) may be accessed by [use association](#).

8 5.5.6 Operator

- 9 1 This document specifies a number of [intrinsic](#) operators (e.g., the arithmetic operators +, −, *, /, and ** with
10 numeric operands and the logical operators [.AND.](#), [.OR.](#), etc. with logical operands). Additional operators
11 can be defined within a program (7.5.5, 15.4.3.4).

12 5.5.7 Companion processors

- 13 1 A [processor](#) has one or more [companion processors](#). A [companion processor](#) can be a mechanism that
14 references and defines such entities by a means other than Fortran (15.6.3), it can be the [Fortran processor](#)
15 itself, or it can be another [Fortran processor](#). If there is more than one [companion processor](#), the means by
16 which the [Fortran processor](#) selects among them are [processor dependent](#).
- 17 2 If a [procedure](#) is defined by means of a [companion processor](#) that is not the [Fortran processor](#) itself, this
18 document refers to the C function that defines the [procedure](#), although the [procedure](#) need not be defined
19 by means of the C programming language.

NOTE

A [companion processor](#) might or might not be a mechanism that conforms to the requirements of ISO/IEC 9899:2018. If it does, 5.3.7 states that a program unit that is defined by means other than Fortran and that initiates normal termination is required to include the effect of executing the C `exit()` function.

For example, a [processor](#) might allow a [procedure](#) defined by some language other than Fortran or C to be invoked if it can be described by a C prototype as defined in ISO/IEC 9899:2018, 6.7.6.3.

6 Lexical tokens and source form

6.1 Processor character set

6.1.1 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 (6.1.2), digits (6.1.3), underscore (6.1.4), special characters (6.1.5), and other characters (6.1.6).

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*.

R601 *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 6.1.2, 6.1.3, 6.1.4, and 6.1.5. However, the style of any graphic is not specified.

6.1.2 Letters

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* (3.21).

NOTE

The following statements are equivalent:

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

6.1.3 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*.

6.1.4 Underscore

R602 *underscore* **is** _

6.1.5 Special characters

- 1 The special characters are shown in Table 6.1.

Table 6.1 — 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.

6.1.6 Other characters

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

6.2 Low-level syntax

6.2.1 Tokens

- 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, **.NIL.**, operator, statement label, delimiter, comma, =, =>, :, ::, ;;, .., ?, or %.

6.2.2 Names

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

R603 *name* **is** *letter* [*alphanumeric-character*] ...

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

NOTE 1

Examples of names:

```

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

```

NOTE 2

The word “name” always denotes this particular syntactic form. The word “identifier” is used where entities can be identified by other syntactic forms or by values; its particular meaning depends on the context in which it is used.

6.2.3 Constants

R604 *constant* **is** *literal-constant*
or *named-constant*

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

R606 *named-constant* **is** *name*

R607 *int-constant* **is** *constant*

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

6.2.4 Operators

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

R1008 *power-op* **is** ******

R1009 *mult-op* **is** *****
or **/**

R1010 *add-op* **is** **+**
or **-**

R1012 *concat-op* **is** **//**

R1014 *rel-op* **is** **.EQ.**
or **.NE.**
or **.LT.**
or **.LE.**
or **.GT.**
or **.GE.**
or **==**
or **/=**

1		or <
2		or <=
3		or >
4		or >=
5	R1019 <i>not-op</i>	is .NOT.
6	R1020 <i>and-op</i>	is .AND.
7	R1021 <i>or-op</i>	is .OR.
8	R1022 <i>equiv-op</i>	is .EQV.
9		or .NEQV.
10	R609 <i>defined-operator</i>	is <i>defined-unary-op</i>
11		or <i>defined-binary-op</i>
12		or <i>extended-intrinsic-op</i>
13	R1004 <i>defined-unary-op</i>	is . letter [letter]
14	R1024 <i>defined-binary-op</i>	is . letter [letter]
15	R610 <i>extended-intrinsic-op</i>	is <i>intrinsic-operator</i>

6.2.5 Statement labels

- 1 A **statement label** provides a means of referring to an individual statement.
- R611 *label* **is** *digit [digit [digit [digit [digit]]]]*
- C603 (R611) 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

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 form¹, may begin with a statement label, but the labels are used only in the following ways.
- The label on a **branch target statement** (11.2) is used to identify that statement as the possible destination of a branch.
 - The label on a **FORMAT statement** (13.2.1) is used to identify that statement as the format specification for a **data transfer statement** (12.6).

¹Fixed form is obsolescent.

- In some forms of the [DO construct \(11.1.7\)](#), the terminal statement of the construct is identified by a label. This syntax is obsolescent.

6.2.6 Delimiters

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

6.3 Source form

6.3.1 Program units, statements, and lines

- 1 A Fortran [program unit](#) is a sequence of one or more [lines](#), organized as Fortran statements, comments, and [INCLUDE lines](#). A [line](#) is a sequence of zero or more characters. [Lines](#) following a [program unit END statement](#) are not part of that [program unit](#). A Fortran [statement](#) is a sequence of one or more complete or partial [lines](#).
- 2 A comment may contain any character that may occur in any [character context](#).
- 3 There are two source forms. The rules in [6.3.2](#) apply only to free form source. The rules in [6.3.3](#) apply only to fixed source form. Free form and fixed form shall not be mixed in the same program unit. The means for specifying the source form of a [program unit](#) are processor dependent.

6.3.2 Free source form

6.3.2.1 Free form line length

- 1 In free source form there are no restrictions on where a statement (or portion of a statement) can appear within a [line](#). A [line](#) may contain zero characters. A [line](#) shall contain at most ten thousand characters.

6.3.2.2 Blank characters in free form

- 1 In free source form blank characters shall not appear within lexical tokens other than in a [character context](#) or in a format specification. Blanks may be inserted freely between tokens to improve readability; for example, blanks may occur between the tokens that form a complex literal constant. A sequence of blank characters outside of a [character context](#) is equivalent to a single blank character.
- 2 A blank shall be used to separate names, constants, or labels from adjacent keywords, names, constants, or labels.

NOTE

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 6.2 — Adjacent keywords where separating blanks are optional

BLOCK DATA	END FILE	END SUBROUTINE
DOUBLE PRECISION	END FORALL	END TEAM

Adjacent keywords where separating blanks are optional (cont.)

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

6.3.2.3 Free form commentary

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

NOTE

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

6.3.2.4 Free form statement continuation

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.

If a non**character 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**.

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.

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

6.3.2.5 Free form statement termination

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

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

6.3.2.6 Free form statements

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

NOTE

No Fortran statement begins with a digit.

- 2 A statement shall not have more than one million characters.

6.3.3 Fixed source form (obsolescent)

6.3.3.1 General

- 1 In fixed source form, there are restrictions on where a statement can 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 Fixed source form is obsolescent.

6.3.3.2 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 follow the last statement of a [program unit](#). Comments have no effect on the interpretation of the [program unit](#).

NOTE

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

6.3.3.3 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

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.

6.3.3.4 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.

6.3.3.5 Fixed form statements

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

6.4 Including source text

Additional text can be incorporated into the source text of a [program unit](#) during processing. This is accomplished with the INCLUDE line, which has the form

INCLUDE [char-literal-constant](#)

The [char-literal-constant](#) shall not have a kind type parameter value that is a [named-constant](#).

An INCLUDE line is not a Fortran statement.

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

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

When an INCLUDE line is resolved, the first included statement [line](#) shall not be a continuation [line](#) and the last included statement [line](#) shall not be continued.

The interpretation of [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

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](#).

7 Types

7.1 Characteristics of types

7.1.1 The concept of type

- 1 Fortran provides an abstract means whereby data can be categorized without relying on a particular physical representation. This abstract means is the concept of type.
- 2 A type has a name, a set of valid values, a means to denote such values (constants), and a set of operations to manipulate the values.

7.1.2 Type classification

- 1 A type is either an [intrinsic type](#) or a nonintrinsic type.
- 2 This document defines five intrinsic types: integer, real, complex, character, and logical.
- 3 A [derived type](#) is one that is defined by a derived-type definition ([7.5.2](#)) or by an intrinsic module. An enum type is one that is defined by an enum type definition ([7.6.1](#)) or by an intrinsic module. An enumeration type is one that is defined by an enumeration type definition ([7.6.2](#)) or by an intrinsic module. A nonintrinsic type name shall be used only where it is accessible ([7.5.2.2](#)). An intrinsic type is always accessible.

7.1.3 Set of values

- 1 For each type, there is a set of valid values. The set of valid values for logical is completely determined by this document. The sets of valid values for integer, character, and real are processor dependent. The set of valid values for complex consists of the set of all the combinations of the values of the real and imaginary parts. The set of valid values for a derived type is as defined in [7.5.8](#). The set of valid values for an enum type is as defined in [7.6.1](#). The set of valid values for an enumeration type is as defined in [7.6.2](#).

7.1.4 Constants

- 1 The syntax for denoting a value indicates the type, type parameters, and the particular value.
- 2 The syntax for literal constants of each intrinsic type is specified in [7.4](#).
- 3 A [structure constructor](#) ([7.5.10](#)) that is a [constant expression](#) ([10.1.12](#)) denotes a scalar constant value of derived type. An enum constructor ([7.6.1](#)) that is a constant expression denotes a scalar constant value of enum type. An enumeration constructor ([7.6.2](#)) that is a constant expression denotes a scalar constant value of enumeration type. An array constructor ([7.8](#)) that is a [constant expression](#) denotes a constant array value of intrinsic or nonintrinsic type.
- 4 A constant value can be named ([8.5.13](#), [8.6.11](#)).

7.1.5 Operations

- 1 For each of the intrinsic types, a set of operations and corresponding operators is defined intrinsically. These are described in [Clause 10](#). The intrinsic set can be augmented with operations and operators

1 defined by functions with the [OPERATOR](#) interface ([15.4.3.2](#)). Operator definitions are described in Clauses
2 [10](#) and [15](#).

3 2 For derived types, there are no intrinsic operations. Operations on derived types can be defined by the
4 program ([7.5.11](#)).

5 3 For an enum or enumeration type, a set of intrinsic operations is defined intrinsically as described in Clause
6 [10](#). The intrinsic set can be augmented with operations and operators defined by the program.

7 7.2 Type parameters

8 1 If a type has type parameters, the set of values, the syntax for denoting the values, and the set of operations
9 on the values of the type depend on the values of the parameters.

10 2 A [type parameter](#) is either a [kind type parameter](#) or a [length type parameter](#). All type parameters are of type
11 integer. A [kind type parameter](#) participates in generic resolution ([15.5.5.2](#)), but a [length type parameter](#)
12 does not.

13 3 Each intrinsic type has a [kind type parameter](#) named KIND. The intrinsic character type has a [length type](#)
14 [parameter](#) named LEN. A derived type can have type parameters.

15 4 A type parameter value can be specified by a type specification ([7.4](#), [7.5.9](#)).

16 R701 *type-param-value* **is** *scalar-int-expr*
17 **or** *
18 **or** :

19 C701 (R701) The *type-param-value* for a kind type parameter shall be a [constant expression](#).

20 C702 (R701) A colon shall not be used as a *type-param-value* except in the declaration of an entity that
21 has the [POINTER](#) or [ALLOCATABLE](#) attribute.

22 5 A colon as a *type-param-value* specifies a [deferred type parameter](#).

23 6 The values of the [deferred type parameters](#) of an object are determined by successful execution of an [AL-](#)
24 [LOCATE](#) statement ([9.7.1](#)), execution of an [intrinsic assignment statement](#) ([10.2.1.3](#)), execution of a [pointer](#)
25 [assignment statement](#) ([10.2.2](#)), or by [argument association](#) ([15.5.2](#)).

NOTE 1

[Deferred type parameters](#) of functions, including function [procedure pointers](#), have no values. Instead, they indicate that those [type parameters](#) of the function result will be determined by execution of the function, if it returns an allocated [allocatable](#) result or an associated pointer result.

26 7 An asterisk as a *type-param-value* specifies that a [length type parameter](#) is an [assumed type parameter](#).
27 It is used for a [dummy argument](#) to assume the type parameter value from the [effective argument](#), for an
28 [associate name](#) in a [SELECT TYPE](#) construct to assume the type parameter value from the corresponding
29 selector, and for a [named constant](#) of type character to assume the character length from the *constant-expr*.

NOTE 2

The value of a [kind type parameter](#) is always known at compile time. Some parameterizations that involve multiple representation forms need to be distinguished at compile time for practical implementation and performance. Examples include the multiple precisions of the intrinsic real type and the possible multiple character sets of the intrinsic character type.

NOTE 2 (cont.)

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.

7.3 Types, type specifiers, and values

7.3.1 Relationship of types and values to objects

- 1 The name of a type serves as a type specifier and can be used to declare objects of that type. A declaration can specify the type of a named object. A data object can be declared explicitly or implicitly. A data object has *attributes* in addition to its type. Clause 8 describes the way in which a data object is declared and how its type and other attributes are specified.
- 2 An array is formed of scalar data of an intrinsic or nonintrinsic type, and has the same type and type parameters as its elements.
- 3 A variable is a data object. The type and type parameters of a variable determine which values that variable can take. Assignment (10.2) provides one means of changing the value of a variable.
- 4 The type of a variable determines the operations that can be used to manipulate the variable.

7.3.2 Type specifiers

7.3.2.1 Type specifier syntax

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

R702 *type-spec* **is** *intrinsic-type-spec*
 or *derived-type-spec*
 or *enum-type-spec*
 or *enumeration-type-spec*

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

R703 *declaration-type-spec* **is** *intrinsic-type-spec*
 or TYPE (*intrinsic-type-spec*)
 or TYPE (*derived-type-spec*)
 or TYPE (*enum-type-spec*)
 or TYPE (*enumeration-type-spec*)
 or CLASS (*derived-type-spec*)
 or CLASS (*)
 or TYPE (*)
 or TYPEOF (*data-ref*)
 or CLASSOF (*data-ref*)

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

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

- 1 C706 (R703) TYPE(*derived-type-spec*) shall not specify an **abstract type** (7.5.7).
- 2 C707 (R702) In TYPE(*intrinsic-type-spec*) the *intrinsic-type-spec* shall not end with a comma.
- 3 C708 An entity declared with the **CLASS** or **CLASSOF** keyword shall be a **dummy argument** or have the
- 4 **ALLOCATABLE** or **POINTER** attribute.
- 5 C709 A TYPEOF or CLASSOF specifier shall appear only in a **type declaration statement** or **component**
- 6 **definition statement**.
- 7 C710 The *data-ref* in a TYPEOF or CLASSOF specifier shall have its type and type parameters previously
- 8 declared or established by the implicit typing rules.
- 9 C711 The *data-ref* in a TYPEOF specifier shall not be unlimited polymorphic or of abstract type.
- 10 C712 The *data-ref* in a CLASSOF specifier shall not be **assumed-type** or of intrinsic type.
- 11 C713 If the *data-ref* in a TYPEOF or CLASSOF specifier has the **OPTIONAL attribute**, it shall not have a
- 12 **deferred** or **assumed type parameter**.
- 13 2 An *intrinsic-type-spec* specifies the named intrinsic type and its type parameter values. A *derived-type-spec*
- 14 specifies the named derived type and its type parameter values. An *enum-type-spec* specifies the named
- 15 enum type. An *enumeration-type-spec* specifies the named enumeration type.
- 16 3 TYPEOF and CLASSOF with a *data-ref* that is not unlimited polymorphic specify the same type and type
- 17 parameter values as the declared type and type parameter values of *data-ref*, except that they specify that a
- 18 type parameter is deferred if it is deferred in *data-ref*. An entity declared with CLASSOF is polymorphic, and
- 19 one declared with TYPEOF is not polymorphic. If a *data-ref* is **CLASS (*)**, CLASSOF (*data-ref*) is equivalent
- 20 to a **CLASS (*)** specifier.

NOTE 1

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

NOTE 2

Note that TYPEOF and CLASSOF declare entities whose type parameters depend on those of the *data-ref*, they are not equivalent to simply repeating the declaration of the *data-ref*. For example, if the *data-ref* has an assumed type parameter, the entities declared have the same values for that type parameter as *data-ref*, they are not assumed (even if they are **dummy arguments**).

21 7.3.2.2 TYPE type specifier

- 22 1 A TYPE type specifier is used to declare entities that are **assumed-type**, or of an intrinsic or nonintrinsic
- 23 type.
- 24 2 A *derived-type-spec*, *enum-type-spec*, or *enumeration-type-spec* in a TYPE type specifier in a **type declaration**
- 25 **statement** shall specify a previously defined type. If the data entity is a function result, the type may be
- 26 specified in the **FUNCTION statement** provided the type is defined within the body of the function or is
- 27 accessible there by use or **host** association. If the type is specified in the **FUNCTION statement** and is defined
- 28 within the body of the function, it is as if the **function result** were declared with that type immediately
- 29 following the definition of the specified type.
- 30 3 An entity that is declared using the TYPE(*) type specifier is **assumed-type** and is an **unlimited polymorphic**
- 31 entity. It is not declared to have a type, and is not considered to have the same **declared type** as any other

entity, including another **unlimited polymorphic** entity. Its **dynamic type** and **type parameters** are assumed from its **effective argument**.

C714 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**.

C715 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**.

C716 An **assumed-type actual argument** that corresponds to an **assumed-rank dummy argument** shall be **assumed-shape** or **assumed-rank**.

7.3.2.3 CLASS type specifier

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

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

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

5 The **dynamic type** of an allocated **allocatable polymorphic** object is the type with which it was allocated. The **dynamic type** of an associated **polymorphic** pointer is the **dynamic type** of its **target**. The **dynamic type** of a nonallocatable nonpointer **polymorphic dummy argument** is the **dynamic type** of its **effective argument**. The **dynamic type** of an unallocated **allocatable** object or a **disassociated** pointer is the same as its **declared type**. The **dynamic type** of an entity identified by an **associate name** (11.1.3) is the **dynamic type** of the selector with which it is associated. The **dynamic type** of an object that is not **polymorphic** is its **declared type**.

7.3.3 Type compatibility

1 A nonpolymorphic entity is **type compatible** only with entities of the same **declared type**, except that an entity of an enum type is also type compatible with an expression of type integer if the expression has a primary that is an enumerator of that enum 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

Given

```

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

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

- 1 2 A **polymorphic allocatable** object may be allocated to be of any type with which it is **type compatible**. A
 2 **polymorphic** pointer or **dummy argument** may, during program execution, be associated with objects with
 3 which it is **type compatible**.

4 7.4 Intrinsic types

5 7.4.1 Classification and specification

- 6 1 Each intrinsic type is classified as a **numeric type** or a nonnumeric type. The **numeric types** are integer,
 7 real, and complex. The nonnumeric intrinsic types are character and logical.

- 8 2 Each intrinsic type has a **kind type parameter** named KIND; this **type parameter** is of type integer with
 9 default kind.

10 R704 *intrinsic-type-spec* **is** *integer-type-spec*
 11 **or** REAL [*kind-selector*]
 12 **or** DOUBLE PRECISION
 13 **or** COMPLEX [*kind-selector*]
 14 **or** CHARACTER [*char-selector*]
 15 **or** LOGICAL [*kind-selector*]

16 R705 *integer-type-spec* **is** INTEGER [*kind-selector*]

17 R706 *kind-selector* **is** ([KIND =] *scalar-int-constant-expr*)

18 C717 (R706) The value of *scalar-int-constant-expr* shall be nonnegative and shall specify a representation
 19 method that exists on the processor.

20 7.4.2 Intrinsic operations on intrinsic types

- 21 1 Intrinsic numeric operations are defined as specified in 10.1.5.2.1 for the numeric intrinsic types. Rela-
 22 tional intrinsic operations are defined as specified in 10.1.5.5 for numeric and character intrinsic types.
 23 The intrinsic concatenation operation is defined as specified in 10.1.5.3 for the character type. Logical
 24 intrinsic operations are defined as specified in 10.1.5.4 for the logical type.

25 7.4.3 Numeric intrinsic types

26 7.4.3.1 Integer type

- 27 1 The set of values for the integer type is a subset of the mathematical integers. The processor shall provide
 28 one or more representation methods that define sets of values for data of type integer. Each such method

- 1 is characterized by a value for the *kind type parameter* *KIND*. The *kind type parameter* of a representation
 2 method is returned by the intrinsic function *KIND* (16.9.118). The decimal exponent range of a represent-
 3 ation method is returned by the intrinsic function *RANGE* (16.9.170). The intrinsic function *SELECTED_-*
 4 *INT_KIND* (16.9.181) returns a kind value based on a specified decimal exponent range requirement. The
 5 integer type includes a zero value, which is considered to be neither negative nor positive. The value of a
 6 signed integer zero is the same as the value of an unsigned integer zero.
- 7 2 The processor shall provide at least one representation method with a decimal exponent range greater
 8 than or equal to 18.
- 9 3 The type specifier for the integer type uses the keyword *INTEGER*.
- 10 4 The keyword *INTEGER* with no *kind-selector* specifies type integer with default kind; the *kind type para-*
 11 *meter* value is equal to *KIND* (0). The decimal exponent range of default integer shall be at least 5.
- 12 5 Any integer value can be represented as a *signed-int-literal-constant*.
- 13 R707 *signed-int-literal-constant* **is** [*sign*] *int-literal-constant*
- 14 R708 *int-literal-constant* **is** *digit-string* [*_ kind-param*]
- 15 R709 *kind-param* **is** *digit-string*
 16 **or** *scalar-int-constant-name*
- 17 R710 *signed-digit-string* **is** [*sign*] *digit-string*
- 18 R711 *digit-string* **is** *digit* [*digit*] ...
- 19 R712 *sign* **is** +
 20 **or** -
- 21 C718 (R709) A *scalar-int-constant-name* shall be a *named constant* of type integer.
- 22 C719 (R709) The value of *kind-param* shall be nonnegative.
- 23 C720 (R708) The value of *kind-param* shall specify a representation method that exists on the processor.
- 24 6 The optional *kind type parameter* following *digit-string* specifies the *kind type parameter* of the integer
 25 constant; if it does not appear, the constant is default integer.
- 26 7 An integer constant is interpreted as a decimal value.

NOTE

Examples of signed integer literal constants are:

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

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

7.4.3.2 Real type

1 The real type has values that approximate the mathematical real numbers. The processor shall provide two or more approximation methods that define sets of values for data of type real. Each such method has a representation method and is characterized by a value for the **kind type parameter** **KIND**. The **kind type parameter** of an approximation method is returned by the intrinsic function **KIND** (16.9.118).

2 The decimal precision, decimal exponent range, and radix of an approximation method are returned by the intrinsic functions **PRECISION** (16.9.162), **RANGE** (16.9.170), and **RADIX** (16.9.166). The intrinsic function **SELECTED_REAL_KIND** (16.9.183) returns a kind value based on specified precision, range, and radix requirements.

NOTE 1

See C.3.1 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 2

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, and

(X < 0.0)

evaluates to false for X = −0.0.

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

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.

R713 *signed-real-literal-constant* **is** [*sign*] *real-literal-constant*

R714 *real-literal-constant* **is** *significand* [*exponent-letter exponent*] [*_ kind-param*]
or *digit-string exponent-letter exponent* [*_ kind-param*]

R715 *significand* **is** *digit-string* . [*digit-string*]
or . *digit-string*

- 1 R716 *exponent-letter* is E
 2 or D
- 3 R717 *exponent* is *signed-digit-string*
- 4 C721 (R714) If both *kind-param* and *exponent-letter* appear, *exponent-letter* shall be E.
- 5 C722 (R714) The value of *kind-param* shall specify an approximation method that exists on the processor.
- 6 7 A real literal constant without a *kind type parameter* is a default real constant if it is without an exponent
 7 part or has exponent letter E, and is a double precision real constant if it has exponent letter D. A real literal
 8 constant written with a *kind type parameter* is a real constant with the specified *kind type parameter*.
- 9 8 The exponent represents the power of ten scaling to be applied to the significand or digit string. The mean-
 10 ing of these constants is as in decimal scientific notation.
- 11 9 The significand may be written with more digits than a processor will use to approximate the value of the
 12 constant.

NOTE 3

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*.

13 7.4.3.3 Complex type

- 14 1 The complex type has values that approximate the mathematical complex numbers. The values of a com-
 15 plex type are ordered pairs of real values. The first real value is called the real part, and the second real
 16 value is called the imaginary part.
- 17 2 Each approximation method used to represent data entities of type real shall be available for both the real
 18 and imaginary parts of a data entity of type complex. The (default integer) *kind type parameter* KIND
 19 for a complex entity specifies for both parts the real approximation method characterized by this kind type
 20 parameter value. The *kind type parameter* of an approximation method is returned by the intrinsic function
 21 *KIND* (16.9.118).
- 22 3 The type specifier for the complex type uses the keyword COMPLEX. There is no keyword for double preci-
 23 sion complex. If the type keyword COMPLEX is used without a *kind type parameter*, the complex type with
 24 default complex kind is specified, the kind value is *KIND* (0.0), and both parts are default real.
- 25 R718 *complex-literal-constant* is (*real-part* , *imag-part*)
- 26 R719 *real-part* is *signed-int-literal-constant*
 27 or *signed-real-literal-constant*
 28 or *named-constant*
- 29 R720 *imag-part* is *signed-int-literal-constant*
 30 or *signed-real-literal-constant*

or *named-constant*

C723 (R718) Each **named constant** in a complex literal constant shall be scalar and 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

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 constant** of type real.

7.4.4 Character type

7.4.4.1 Character sets

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

2 The processor shall provide one or more representation methods that define sets of values for data of type character. Each such method is characterized by a value for the (default integer) **kind type parameter** KIND. The **kind type parameter** of a representation method is returned by the intrinsic function **KIND** (16.9.118). The intrinsic function **SELECTED_CHAR_KIND** (16.9.180) returns a kind value based on the name of a character type. Any character of a particular representation method representable in the processor may occur in a character string of that representation method.

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

7.4.4.2 Character type specifier

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

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

3 The default character kind shall support a character set that includes the characters in the Fortran character set (6.1). The processor may support additional character sets by supplying nondefault character kinds.

The characters available in nondefault character kinds are not specified by this document, except that one character in each nondefault character set shall be designated as a blank character to be used as a padding character.

R721 *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*])

R722 *length-selector* is ([LEN =] *type-param-value*)
 or * *char-length* [,]

4 The “* *char-length*” (and optional comma) syntax in a *length-selector* is obsolescent.

R723 *char-length* is (*type-param-value*)
 or *int-literal-constant*

C724 (R721) The value of *scalar-int-constant-expr* shall be nonnegative and shall specify a representation method that exists on the processor.

C725 (R723) The *int-literal-constant* shall not include a *kind-param*.

C726 (R721 R722 R723) A *type-param-value* of * shall be used only

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

C727 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.

C728 A function name declared with an asterisk *type-param-value* shall not be an array, a pointer, *elemental*, or pure. A function name declared with an asterisk *type-param-value* shall not have the *RECURSIVE* attribute.

C729 (R722) The optional comma in a *length-selector* is permitted only in a *declaration-type-spec* in a *type-declaration-stmt*.

C730 (R722) The optional comma in a *length-selector* is permitted only if no double-colon separator appears in the *type-declaration-stmt*.

C731 (R721) The length specified for a character *statement function*¹ or for a *statement function dummy argument* of type character shall be a *constant expression*.

¹Statement functions are obsolescent.

5 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.

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

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

7.4.4.3 Character literal constant

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

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

C732 (R724) The value of *kind-param* shall specify a representation method that exists on the processor.

2 The optional *kind type parameter* preceding the leading delimiter specifies the *kind type parameter* of the character constant; if it does not appear, the constant is default character.

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 form², 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

²Fixed source form is obsolescent.

- 1 is represented by two consecutive quotation marks (without intervening blanks) and the two quotation
 2 marks are counted as one character.
- 3 6 A zero-length character literal constant is represented by two consecutive apostrophes (without interven-
 4 ing blanks) or two consecutive quotation marks (without intervening blanks) outside of a [character context](#).

NOTE 1

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 2

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

```
NIHONGO_ '彼女なしで何もできない。'
```

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

5 **7.4.4.4 Collating sequence**

- 6 1 The processor defines a [collating sequence](#) for the character set of each kind of character. The [collating](#)
 7 [sequence](#) is an isomorphism between the character set and the set of integers $\{I : 0 \leq I < N\}$, where N is
 8 the number of characters in the set. The intrinsic functions [CHAR](#) (16.9.52) and [ICHAR](#) (16.9.105) provide
 9 conversions between the characters and the integers according to this mapping.

NOTE 1

For example:

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

returns the integer value of the character 'X' according to the [collating sequence](#) of the processor.

- 10 2 The [collating sequence](#) of the default character kind shall satisfy the following constraints.
- 11 • [ICHAR](#) ('A') < [ICHAR](#) ('B') < ... < [ICHAR](#) ('Z') for the twenty-six upper-case letters.
 - 12 • [ICHAR](#) ('0') < [ICHAR](#) ('1') < ... < [ICHAR](#) ('9') for the ten digits.
 - 13 • [ICHAR](#) (' ') < [ICHAR](#) ('0') < [ICHAR](#) ('9') < [ICHAR](#) ('A') or
 - 14 [ICHAR](#) (' ') < [ICHAR](#) ('A') < [ICHAR](#) ('Z') < [ICHAR](#) ('0').
 - 15 • [ICHAR](#) ('a') < [ICHAR](#) ('b') < ... < [ICHAR](#) ('z') for the twenty-six lower-case letters.
 - 16 • [ICHAR](#) (' ') < [ICHAR](#) ('0') < [ICHAR](#) ('9') < [ICHAR](#) ('a') or
 - 17 [ICHAR](#) (' ') < [ICHAR](#) ('a') < [ICHAR](#) ('z') < [ICHAR](#) ('0').
- 18 3 There are no constraints on the location of any other character in the [collating sequence](#), nor is there any
 19 specified [collating sequence](#) relationship between the upper-case and lower-case letters.

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

NOTE 2

The intrinsic functions [ACHAR \(16.9.3\)](#) and [IACHAR \(16.9.98\)](#) provide conversions between characters and corresponding integer values according to the ASCII [collating sequence](#).

4 5 The intrinsic functions `LGT`, `LGE`, `LLE`, and `LLT` (16.9.124-16.9.127) provide comparisons between strings
5 based on the ASCII collating sequence. International portability is guaranteed if the set of characters used
6 is limited to the Fortran character set (6.1).

7 7.4.5 Logical type

8 1 The logical type has two values, which represent true and false.

9 2 The processor shall provide one or more representation methods for data of type logical. Each such method
10 is characterized by a value for the (default integer) **kind type parameter** **KIND**. The **kind type parameter** of
11 a representation method is returned by the intrinsic function **KIND** (16.9.118).

12 3 The type specifier for the logical type uses the keyword LOGICAL.

13 4 The keyword LOGICAL with no *kind-selector* specifies type logical with default kind; the *kind type para-*
14 *meter* value is equal to KIND (.FALSE.).

[illegible]

17 C733 (R725) The value of *kind-param* shall specify a representation method that exists on the processor.

18 5 The optional **kind type parameter** specifies the **kind type parameter** of the logical constant; if it does not
19 appear, the constant has the default logical kind.

20 7.5 Derived types

21 7.5.1 Derived type concepts

22 1 Additional types can be derived from the intrinsic types and other derived types. A type definition defines
23 the name of the type and the names and **attributes** of its components and **type-bound procedures**.

24 2 A derived type can be parameterized by one or more **type parameters**, each of which is defined to be either
25 a **kind** or **length** type parameter and can have a default value.

26 3 The **ultimate components** of a derived type are the components that are of intrinsic type or have the **AL-**
27 **LOCATABLE** or **POINTER attribute**, plus the **ultimate components** of the components that are of derived
28 type and have neither the **ALLOCATABLE** nor **POINTER attribute**.

29 4 The **direct components** of a derived type are the components of that type, plus the **direct components** of
30 the components that are of derived type and have neither the **ALLOCATABLE** nor **POINTER** attribute.

31 5 The **potential subobject components** of a derived type are the nonpointer components of that type together
32 with the **potential subobject components** of the nonpointer components that are of derived type. This
33 includes all the components that could be a subobject of an object of the type (9.4.2).

- 1 6 The **components**, **direct components**, **potential subobject components**, and **ultimate components** of an ob-
 2 ject of derived type are the **components**, **direct components**, **potential subobject components**, and **ultimate**
 3 **components** of its type, respectively.
- 4 7 By default, no **storage sequence** is implied by the order of the component definitions. However, a storage
 5 sequence is implied for a **sequence type** (7.5.2.3). If the derived type has the **BIND attribute**, the **storage**
 6 **sequence** is that required by the **companion processor** (5.5.7, 18.3.4).
- 7 8 A scalar entity of derived type is a **structure**. If a derived type has the **SEQUENCE attribute**, a scalar entity
 8 of the type is a **sequence structure**.

NOTE

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

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

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

9 **7.5.2 Derived-type definition**

10 **7.5.2.1 Syntax of a derived-type definition**

- 11 R726 *derived-type-def* **is** *derived-type-stmt*
 12 [*type-param-def-stmt*] ...
 13 [*private-or-sequence*] ...
 14 [*component-part*]
 15 [*type-bound-procedure-part*]
 16 *end-type-stmt*
- 17 R727 *derived-type-stmt* **is** TYPE [[, *type-attr-spec-list*] ::] *type-name* ■
 18 ■ [(*type-param-name-list*)]
- 19 R728 *type-attr-spec* **is** ABSTRACT
 20 **or** *access-spec*
 21 **or** BIND (C)
 22 **or** EXTENDS (*parent-type-name*)
- 23 C734 (R727) A derived type *type-name* shall not be **DOUBLEPRECISION** or the same as the name of any
 24 intrinsic type defined in this document.
- 25 C735 (R727) The same *type-attr-spec* shall not appear more than once in a given *derived-type-stmt*.
- 26 C736 The same *type-param-name* shall not appear more than once in a given *derived-type-stmt*.
- 27 C737 (R728) A *parent-type-name* shall be the name of a previously defined **extensible type** (7.5.7).
- 28 C738 (R726) If the type definition contains or **inherits** (7.5.7.2) a deferred **type-bound procedure** (7.5.5),
 29 **ABSTRACT** shall appear.

- 1 C739 (R726) If **ABSTRACT** appears, the type shall be **extensible**.
- 2 C740 (R726) If **EXTENDS** appears, **SEQUENCE** shall not appear.
- 3 C741 (R726) If **EXTENDS** appears and the type being defined has a **coarray potential subobject component**, its **parent type** shall have a **coarray potential subobject component**.
- 4
- 5 C742 (R726) If **EXTENDS** appears and the type being defined has a **potential subobject component** of
- 6 type **EVENT_TYPE**, **LOCK_TYPE**, or **NOTIFY_TYPE** from the intrinsic module **ISO_FORTRAN_ENV**,
- 7 its **parent type** shall be **EVENT_TYPE**, **LOCK_TYPE**, or **NOTIFY_TYPE**, or have a **potential subobject**
- 8 **component** of type **EVENT_TYPE**, **LOCK_TYPE**, or **NOTIFY_TYPE**.
- 9 R729 *private-or-sequence* **is** *private-components-stmt*
- 10 **or** *sequence-stmt*
- 11 C743 (R726) The same *private-or-sequence* shall not appear more than once in a given *derived-type-def*.
- 12 R730 *end-type-stmt* **is** **END TYPE** [*type-name*]
- 13 C744 (R730) If **END TYPE** is followed by a *type-name*, the *type-name* shall be the same as that in the
- 14 corresponding *derived-type-stmt*.
- 15 1 Derived types with the **BIND attribute** are subject to additional constraints as specified in 18.3.4.

NOTE

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
```

16 **7.5.2.2 Accessibility**

- 17 1 The accessibility of a type name is determined as specified in 8.5.2. The accessibility of a type name does
- 18 not affect, and is not affected by, the accessibility of its **components** and **type-bound procedures**.
- 19 2 If a derived type is defined in the scoping unit of a module, and its name is private in that module, then the
- 20 type name, and thus the structure constructor (7.5.10) for the type, are accessible only within that module
- 21 and its **descendants**.

NOTE

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**.

22 **7.5.2.3 Sequence type**

- 23 R731 *sequence-stmt* **is** **SEQUENCE**

1 C745 (R726) If SEQUENCE appears, the type shall have at least one component, each data component
 2 shall be declared to be of an intrinsic type or of a [sequence type](#), the derived type shall not have any
 3 [type parameter](#), and a [type-bound-procedure-part](#) shall not appear.

4 1 If the SEQUENCE statement appears, the type has the [SEQUENCE attribute](#) and is a [sequence type](#). The
 5 order of the component definitions in a [sequence type](#) specifies a [storage sequence](#) for objects of that type.
 6 The type is a [numeric sequence type](#) if there are no [pointer](#) or [allocatable components](#), and each [component](#)
 7 is default integer, default real, double precision real, default complex, default logical, or of [numeric sequence](#)
 8 [type](#). The type is a [character sequence type](#) if there are no [pointer](#) or [allocatable](#) components, and each
 9 component is default character or of [character sequence type](#).

NOTE 1

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 2

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 sequence of data values in namelist input data, and
- 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.

10 7.5.2.4 Determination of derived types

- 11 1 Derived-type definitions with the same type name may appear in different [scoping units](#), in which case they
 12 might be independent and describe different derived types or they might describe the same type.
- 13 2 Two data entities have the same type if they are declared with reference to the same derived-type defin-
 14 ition. Data entities also have the same type if they are declared with reference to different derived-type
 15 definitions that specify the same type name, all have the [SEQUENCE attribute](#) or all have the [BIND attrib-](#)
 16 [ute](#), have no components with [PRIVATE](#) accessibility, and have components that agree in order, name, and
 17 attributes. Otherwise, they are of different derived types. A data entity declared using a type with the [SE-](#)
 18 [QUENCE attribute](#) or with the [BIND attribute](#) is not of the same type as an entity of a type that has any
 19 components that are [PRIVATE](#).

NOTE 1

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

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

NOTE 1 (cont.)

```

TYPE (POINT) :: X1
CALL SUB (X1)
...
CONTAINS
  SUBROUTINE SUB (A)
    TYPE (POINT) :: A
    ...
  END SUBROUTINE SUB

```

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

NOTE 2

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

NOTE 3

The requirement that the two types have the same name applies to the *type-names* in the respective derived type definitions, not to local names introduced via renaming in [USE statements](#).

1 **7.5.3 Derived-type parameters**2 **7.5.3.1 Type parameter definition statement**

3 R732 *type-param-def-stmt* is [integer-type-spec](#), [type-param-attr-spec](#) :: ■
 4 ■ [type-param-decl-list](#)

5 R733 *type-param-decl* is [type-param-name](#) [= [scalar-int-constant-expr](#)]

6 C746 (R732) A [type-param-name](#) in a [type-param-def-stmt](#) in a [derived-type-def](#) shall be one of the *type-*
 7 *param-names* in the [derived-type-stmt](#) of that [derived-type-def](#).

8 C747 (R732) Each [type-param-name](#) in the [derived-type-stmt](#) in a [derived-type-def](#) shall appear exactly

once as a *type-param-name* in a *type-param-def-stmt* in that *derived-type-def*.

R734 *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 (10.2.1.3) to a value of the same kind as the type parameter.

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

NOTE

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 potentially nondefault kind for d.
  REAL (k) :: element (d, d)
END TYPE
```

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

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

7.5.3.2 Type parameter order

1 *Type parameter order* is an ordering of the type parameters of a derived type; it is used for derived-type specifiers.

2 The *type parameter order* of a nonextended type is the order of the *type-param-name-list* in the derived-type definition. The *type parameter order* of an *extended type* (7.5.7) consists of the *type parameter order* of its *parent type* followed by any additional type parameters in the order of the *type-param-name-list* in the derived-type definition.

NOTE

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
```


NOTE (cont.)

the type parameter order for type t_1 is k_1 then k_2 , and the type parameter order for type t_2 is k_1 then k_2 then k_3 .

7.5.4 Components

7.5.4.1 Component definition statement

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

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

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

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

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

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

C748 (R737) No *component-attr-spec* shall appear more than once in a given *component-def-stmt*.

C749 (R737) If neither the POINTER nor the ALLOCATABLE attribute is specified, the *declaration-type-spec* in the *component-def-stmt* shall specify an intrinsic type, or a previously defined derived, enum, or enumeration type.

C750 (R737) If the POINTER or ALLOCATABLE attribute is specified, each *component-array-spec* shall be a *deferred-shape-spec-list*.

C751 (R737) If a *coarray-spec* appears, it shall be a *deferred-coshape-spec-list* and the component shall have the ALLOCATABLE attribute.

C752 (R737) If a *coarray-spec* appears, the component shall not be of type C_PTR or C_FUNPTR from the intrinsic module ISO_C_BINDING (18.2), or of type TEAM_TYPE from the intrinsic module ISO_FORTRAN_ENV (16.10.2).

C753 (R737) If neither the POINTER nor the ALLOCATABLE attribute is specified, each *component-array-spec* shall be an *explicit-shape-spec-list*.

C754 (R740) Each bound in the *explicit-shape-spec* shall be a component specification expression.

C755 (R737) A component shall not have both the ALLOCATABLE and POINTER attributes.

C756 (R737) If the CONTIGUOUS attribute is specified, the component shall be an array with the POINTER

attribute.

C757 (R739) The * *char-length* option is permitted only if the component is of type character.

C758 (R736) Each *type-param-value* within a *component-def-stmt* shall be a colon or a *component specification expression*.

NOTE 1

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*.

NOTE 2

A data component whose type has a *coarray potential subobject component* cannot be a *coarray* or a *pointer*, see constraint C825.

R741 *proc-component-def-stmt* is PROCEDURE ([*proc-interface*]), ■
■ *proc-component-attr-spec-list* :: *proc-decl-list*

NOTE 3

See 15.4.3.6 for definitions of *proc-interface* and *proc-decl*.

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

C759 The same *proc-component-attr-spec* shall not appear more than once in a given *proc-component-def-stmt*.

C760 (R741) POINTER shall appear in each *proc-component-attr-spec-list*.

C761 (R741) If the *procedure pointer* component has an *implicit interface* or has no arguments, NOPASS shall be specified.

C762 (R741) If PASS (*arg-name*) appears, the *interface* of the *procedure pointer* component shall have a *dummy argument* named *arg-name*.

C763 (R741) 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 can 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* can be specified or overridden for a component by the appearance of a *component-array-spec* in its *component-decl*, and the *CODIMENSION attribute* can be specified or overridden for a component by the appearance of a *coarray-spec* in its *component-decl*.

7.5.4.2 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 (8.5.8.2), the *array bounds*; otherwise, the *component-*

- 1 [array-spec](#) in the DIMENSION clause specifies the array [rank](#), and if the array is explicit shape, the [array](#)
2 [bounds](#).

NOTE 1

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 2

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 3

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

3 7.5.4.3 Coarray components

- 4 1 A data component is a [coarray](#) if its [component-decl](#) contains a [coarray-spec](#) or its [data-component-def-stmt](#)
5 contains a CODIMENSION clause. If the [component-decl](#) contains a [coarray-spec](#) it specifies the [corank](#);
6 otherwise, the [coarray-spec](#) in the CODIMENSION clause specifies the [corank](#).

NOTE

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 a [coarray](#) or a [pointer](#).

7 7.5.4.4 Pointer components

- 8 1 A data component is a [data pointer](#) (5.4.9) if its [component-attr-spec-list](#) contains the **POINTER** keyword.
9 A [procedure pointer](#) component has the **POINTER** keyword in its [proc-component-attr-spec-list](#).

NOTE

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

NOTE (cont.)

```

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

```

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

7.5.4.5 The passed-object dummy argument

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

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](#).

C764 The [passed-object dummy argument](#) shall be a scalar, nonpointer, nonallocatable [dummy data object](#) with the same [declared type](#) as the type being defined; all of its [length type parameters](#) shall be assumed; it shall be [polymorphic](#) ([7.3.2.3](#)) if and only if the type being defined is [extensible](#) ([7.5.7](#)). It shall not have the [VALUE attribute](#).

NOTE

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

7.5.4.6 Default initialization for components

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](#).

```

R743  component-initialization  is  = constant-expr
                                     or => null-init
                                     or => initial-data-target

```

```

R744  initial-data-target      is  designator

```

- 1 C765 If *component-initialization* appears in a *data-component-def-stmt*, a double-colon separator shall
2 appear before the *component-decl-list*.
- 3 C766 (R737) If *component-initialization* appears, every type parameter and *array bound* of the compon-
4 ent shall be a colon or *constant expression*.
- 5 C767 (R737) If *=>* appears in *component-initialization*, *POINTER* shall appear in the *component-attr-*
6 *spec-list*. If *=* appears in *component-initialization*, neither *POINTER* nor *ALLOCATABLE* shall appear
7 in the *component-attr-spec-list*.
- 8 C768 If *initial-data-target* appears in a *component-initialization* in a *component-decl*, *component-name*
9 shall be *data-pointer-initialization* compatible with it.
- 10 C769 A *designator* that is an *initial-data-target* shall designate a nonallocatable, *noncoindexed* variable
11 that has the *TARGET* and *SAVE* attributes and does not have a *vector subscript*. Every subscript,
12 section subscript, substring starting point, and substring ending point in *designator* shall be a *con-*
13 *stant expression*.
- 14 3 If *null-init* appears for a pointer component, that component in any object of the type has an initial associ-
15 ation status of *disassociated* (3.57) or becomes *disassociated* as specified in 19.5.2.4.
- 16 4 If *initial-data-target* appears for a *data pointer* component, that component in any object of the type is
17 initially associated with the *target* or becomes associated with the *target* as specified in 19.5.2.3.
- 18 5 If *initial-proc-target* (15.4.3.6) appears in *proc-decl* for a *procedure pointer* component, that component
19 in any object of the type is initially associated with the *target* or becomes associated with the *target* as
20 specified in 19.5.2.3.
- 21 6 If *constant-expr* appears for a nonpointer component, that component in any object of the type is initially
22 defined (19.6.3) or becomes defined as specified in 19.6.5 with the value determined from *constant-expr*.
23 If necessary, the value is converted according to the rules of intrinsic assignment (10.2.1.3) to a value that
24 agrees in type, type parameters, and shape with the component. If the component is of a type for which
25 *default initialization* is specified for a component, the *default initialization* specified by *constant-expr* over-
26 rides the *default initialization* specified for that component. When one *initialization* overrides another it
27 is as if only the overriding *initialization* were specified (see NOTE 2). *Explicit initialization* in a *type de-*
28 *claration statement* (8.2) overrides *default initialization* (see NOTE 1). Unlike *explicit initialization*, *default*
29 *initialization* does not imply that the object has the *SAVE attribute*.
- 30 7 A *subcomponent* (9.4.2) is *default-initialized* if the type of the object of which it is a component specifies *de-*
31 *fault initialization* for that component, and the *subcomponent* is not a subobject of an object that is *default-*
32 *initialized* or *explicitly initialized*.
- 33 8 A type has *default initialization* if *component-initialization* is specified for any *direct component* of the type.
34 An object has *default initialization* if it is of a type that has *default initialization*.

NOTE 1

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*:

NOTE 1 (cont.)

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

NOTE 2

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 1:

```
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 3

Arrays of structures can be declared with elements that are partially or totally initialized by default. Continuing the example of NOTE 2:

```
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

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

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

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

NOTE 5

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
```

7.5.4.7 Component order

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

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

Given the same type definitions as in 7.5.3.2, NOTE, the **component order** of type T1 is just A (there is only one component), and the **component order** of type T2 is A then FLAG. The **parent component** (T1) does not participate in the **component order**.

7.5.4.8 Component accessibility

R745 *private-components-stmt* **is** PRIVATE

C770 (R745) A *private-components-stmt* is permitted only if the type definition is within the specification part of a module.

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 can be explicitly declared by an *access-spec*; otherwise its accessibility is the default for the type definition in which it is declared.

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

NOTE 1

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

NOTE 2

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 of public and private type names with public and private components.

NOTE 3

An example of a public type with private components is:

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

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

NOTE 4

An example that uses an individual component *access-spec* to override the default accessibility is:

```
TYPE MIXED
    PRIVATE
    INTEGER :: I
```


NOTE 4 (cont.)

```

    INTEGER, PUBLIC :: J
    END TYPE MIXED

    TYPE (MIXED) :: M

```

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

7.5.5 Type-bound procedures

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

[*binding-private-stmt*]

[*type-bound-proc-binding*] ...

R747 *binding-private-stmt* **is** PRIVATE

C771 (R746) A *binding-private-stmt* is permitted only if the type definition is within the specification part of a module.

R748 *type-bound-proc-binding* **is** *type-bound-procedure-stmt*
or *type-bound-generic-stmt*
or *final-procedure-stmt*

R749 *type-bound-procedure-stmt* **is** PROCEDURE [[, *binding-attr-list*] ::] *type-bound-proc-decl-list*
or PROCEDURE (*interface-name*), *binding-attr-list* :: ■
 ■ *binding-name-list*

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

C772 (R749) If => *procedure-name* appears in a *type-bound-proc-decl*, the double-colon separator shall appear.

C773 (R750) The *procedure-name* shall be the name of an accessible [module procedure](#) or an [external procedure](#) that has an [explicit interface](#).

C774 A *binding-name* in a *type-bound-proc-decl* in a derived type definition shall not be the same as any other *binding-name* within that derived type definition.

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

R751 *type-bound-generic-stmt* **is** GENERIC [, *access-spec*] :: *generic-spec* => *binding-name-list*

C775 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.

C776 (R751) Each *binding-name* in *binding-name-list* shall be the name of a specific [binding](#) of the type.

C777 A *binding-name* in a type-bound GENERIC statement shall not specify a specific [binding](#) that was [inherited](#) or specified previously for the same generic identifier in that derived type definition.

C778 (R751) If *generic-spec* is not *generic-name*, each of its specific [bindings](#) shall have a [passed-object](#)

- 1 dummy argument (7.5.4.5).
- 2 C779 (R751) If *generic-spec* is OPERATOR (*defined-operator*), the interface of each binding shall be as
3 specified in 15.4.3.4.2.
- 4 C780 (R751) If *generic-spec* is ASSIGNMENT (=), the interface of each binding shall be as specified in
5 15.4.3.4.3.
- 6 C781 (R751) If *generic-spec* is *defined-io-generic-spec*, the interface of each binding shall be as specified
7 in 12.6.4.8. The type of the `dtv` argument shall be *type-name*.
- 8 R752 *binding-attr* is *access-spec*
9 or DEFERRED
10 or NON_OVERRIDABLE
11 or NOPASS
12 or PASS [(*arg-name*)]
- 13 C782 (R752) The same *binding-attr* shall not appear more than once in a given *binding-attr-list*.
- 14 C783 (R749) If the interface of the binding has no dummy argument of the type being defined, NOPASS
15 shall appear.
- 16 C784 (R749) If PASS (*arg-name*) appears, the interface of the binding shall have a dummy argument
17 named *arg-name*.
- 18 C785 (R752) PASS and NOPASS shall not both appear in the same *binding-attr-list*.
- 19 C786 (R752) NON_OVERRIDABLE and DEFERRED shall not both appear in the same *binding-attr-list*.
- 20 C787 (R752) DEFERRED shall appear if and only if *interface-name* appears.
- 21 C788 (R749) An overriding binding (7.5.7.3) shall have the DEFERRED attribute only if the binding it
22 overrides is deferred.
- 23 C789 (R749) A binding shall not override an inherited binding (7.5.7.2) that has the NON_OVERRIDABLE
24 attribute.
- 25 2 A type-bound procedure statement declares one or more specific type-bound procedures. A specific type-
26 bound procedure can have a passed-object dummy argument (7.5.4.5). A type-bound procedure with the
27 DEFERRED attribute is a deferred type-bound procedure. The DEFERRED keyword shall appear only in
28 the definition of an abstract type.
- 29 3 A GENERIC statement declares a generic type-bound procedure, which is a type-bound generic interface
30 for its specific type-bound procedures.
- 31 4 A binding of a type is a type-bound procedure (specific or generic), a generic type-bound interface, or a final
32 subroutine. These are referred to as specific bindings, generic bindings, and final bindings respectively.
- 33 5 A type-bound procedure can be identified by a binding name in the scope of the type definition. This name
34 is the *binding-name* for a specific type-bound procedure, and the *generic-name* for a generic binding whose
35 *generic-spec* is *generic-name*. A final binding, or a generic binding whose *generic-spec* is not *generic-name*,
36 has no binding name.
- 37 6 The interface of a specific type-bound procedure is that of the procedure specified by *procedure-name* or
38 the interface specified by *interface-name*.

- 1 7 The same *generic-spec* may be used in several GENERIC statements within a single derived-type definition.
 2 Each additional GENERIC statement with the same *generic-spec* extends the *generic interface*.

NOTE 1

Unlike the situation with generic procedure names, a generic *type-bound procedure* name is not permitted to be the same as a specific *type-bound procedure* name in the same type (19.3).

- 3 8 The default accessibility for the *type-bound procedures* of a type is private if the type definition contains
 4 a *binding-private-stmt*, and public otherwise. The accessibility of a *type-bound procedure* can be explicitly
 5 declared by an *access-spec*; otherwise its accessibility is the default for the type definition in which it is
 6 declared.

- 7 9 A public *type-bound procedure* is accessible via any accessible object of the type. A private *type-bound*
 8 *procedure* is accessible only within the module containing the type definition, and within its *descendants*.

NOTE 2

The accessibility of a *type-bound procedure* is not affected by a *PRIVATE statement* in the *component-part*; the accessibility of a component is not affected by a *PRIVATE statement* in the *type-bound-procedure-part*.

NOTE 3

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
```

9 7.5.6 Final subroutines

10 7.5.6.1 FINAL statement

11 R753 *final-procedure-stmt* is FINAL [::] *final-subroutine-name-list*

12 C790 (R753) A *final-subroutine-name* shall be the name of a module procedure with exactly one *dummy*
 13 *argument*. That argument shall be nonoptional and shall be a noncoarray, nonpointer, nonallocat-
 14 able, nonpolymorphic variable of the derived type being defined. All length type parameters of the
 15 *dummy argument* shall be assumed. The *dummy argument* shall not have the *INTENT (OUT)* or
 16 *VALUE* attribute.

17 C791 (R753) A *final-subroutine-name* shall not be one previously specified as a *final subroutine* for that
 18 type.

19 C792 (R753) A *final subroutine* shall not have a *dummy argument* with the same *kind type parameters*
 20 and *rank* as the *dummy argument* of another *final subroutine* of that type.

21 C793 (R753) If a *final subroutine* has an *assumed-rank dummy argument*, no other *final subroutine* of
 22 that type shall have a *dummy argument* with the same *kind type parameters*.

- 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 (7.5.6.2).
- 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 1

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 2

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.

6 7.5.6.2 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, or a **final subroutine** whose **dummy argument** is **assumed-rank** with the same **kind type parameters** as the entity being finalized, it is called with the entity as an **actual argument**. Otherwise, no subroutine is called at this point.
 - (2) All nonallocatable **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 7.5.6.3, 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.

NOTE

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** in the **current team**.

24 7.5.6.3 When finalization occurs

- 1 When an **intrinsic assignment statement** is executed (10.2.1.3), 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, or has an allocated allocatable subobject, 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**. If an error condition occurs during deallocation, it is processor dependent whether finalization occurs.

- 1 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** (19.6.6, item (3)).
- 2
- 3
- 4 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** (19.6.6, item (23)).
- 5
- 6 5 If an executable construct references a nonpointer function, the result is finalized after execution of the innermost executable construct containing the reference.
- 7
- 8 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**.
- 9
- 10 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**.
- 11
- 12
- 13
- 14
- 15 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.
- 16
- 17

NOTE

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

7.5.6.4 Entities that are not finalized

- 1 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.
- 20

NOTE

A nonpointer, nonallocatable object that has the **SAVE attribute** is never finalized as a direct consequence of the execution of a **RETURN** or **END statement**.

7.5.7 Type extension**7.5.7.1 Extensible, extended, and abstract types**

- 23 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**.
- 24
- 25 2 A type with the **EXTENDS attribute** is an **extended type**; its **parent type** is the type named in the **EXTENDS type-attr-spec**.
- 26

NOTE 1

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

- 27 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**.
- 28
- 29 4 An **abstract type** is a type that has the **ABSTRACT attribute**.

NOTE 2

The **DEFERRED** attribute (7.5.5) defers the implementation of a **type-bound procedure** to **extensions** of the type; it can appear only in an **abstract type**. The **dynamic type** of an object cannot be **abstract**; therefore, a deferred **type-bound procedure** cannot be invoked. An **extension** of an **abstract type** need not be **abstract** if it has no deferred **type-bound procedures**. A short example of an **abstract type** is:

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

For a more elaborate example see C.3.4.

7.5.7.2 Inheritance

- 1 An **extended type** includes all of the type parameters, all of the components, and the nonoverridden (7.5.7.3) **type-bound procedures** of its **parent type**. These are **inherited** by the **extended type** from the **parent type**. They retain all of the attributes that they had in the **parent type**. Additional type parameters, components, and procedure **bindings** may be declared in the derived-type definition of the **extended type**.

NOTE 1

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 2

An extensible derived type is not required to have any components, **bindings**, or parameters; an **extended type** is not required to have more components, **bindings**, or parameters than its **parent type**.

- 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. If the **extended type** is defined in a module, the **parent component** has the accessibility of the **parent type** in the module in which the **parent type** was defined. Components of the **parent component** are **inheritance associated** (19.5.4) with the corresponding components **inherited** from the **parent type**. An ancestor component of a type is the **parent component** of the type or an ancestor component of the **parent component**.
- 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 15.4.3.4.5.

NOTE 3

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

NOTE 4

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
```

7.5.7.3 Type-bound procedure overriding

- 1 If a specific **type-bound procedure** specified in a type definition has the same **binding name** as an accessible **type-bound procedure** from the **parent type** then the **binding** specified in the type definition overrides the one from the **parent type**.
- 2 The overriding and overridden **type-bound procedures** shall satisfy the following conditions.
 - Either both shall have a **passed-object dummy argument** or neither shall.
 - If the overridden **type-bound procedure** is **pure** then the overriding one shall also be **pure**.
 - If the overridden **type-bound procedure** is **simple** then the overriding one shall also be **simple**.
 - Either both shall be **elemental** or neither shall.
 - They shall have the same number of **dummy arguments**.
 - **Passed-object dummy arguments**, if any, shall correspond by name and position.
 - **Dummy arguments** that correspond by position shall have the same names and **characteristics**, except for the type of the **passed-object dummy arguments**.
 - Either both shall be subroutines or both shall be functions having the same result **characteristics** (15.3.3).
 - If the overridden **type-bound procedure** is **PUBLIC** then the overriding one shall not be **PRIVATE**.
- 3 A **binding** of a type and a **binding** of an **extension** of that type correspond if the latter **binding** is the same **binding** as the former, overrides a corresponding **binding**, or is an **inherited** corresponding **binding**.

NOTE

The following is an example of procedure overriding, expanding on the example in 7.5.5, NOTE 3.

```
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
```

7.5.8 Derived-type values

- 1 The component value of
 - a pointer component is its **pointer association**,
 - an **allocatable** component is its allocation status and, if it is allocated, its **dynamic type** and type parameters, **bounds** and value, and
 - a nonpointer nonallocatable component is its value.
- 2 The set of values of a particular derived type consists of all possible sequences of the component values of its components.

7.5.9 Derived-type specifier

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

R754 *derived-type-spec* is *type-name* [(*type-param-spec-list*)]

R755 *type-param-spec* is [*keyword* =] *type-param-value*

C794 (R754) *type-name* shall be the name of an accessible derived type.

C795 (R754) *type-param-spec-list* shall appear only if the type is parameterized.

C796 (R754) There shall be at most one *type-param-spec* corresponding to each parameter of the type. If a type parameter does not have a default value, there shall be a *type-param-spec* corresponding to that type parameter.

C797 (R755) The *keyword*= shall not be omitted from a *type-param-spec* unless the *keyword*= has been omitted from each preceding *type-param-spec* in the *type-param-spec-list*.

C798 (R755) Each *keyword* shall be the name of a parameter of the type.

C799 (R755) An asterisk shall not be used as a *type-param-value* in a *type-param-spec* except in the declaration of a *dummy argument* or *associate name* or in the allocation of a *dummy argument*.

2 Type parameter values that do not have *type parameter keywords* specified correspond to type parameters in type parameter order (7.5.3.2). If a *type parameter keyword* appears, the value corresponds to the type parameter named by the keyword. If necessary, the value is converted according to the rules of intrinsic assignment (10.2.1.3) to a value of the same kind as the type parameter.

3 The value of a type parameter for which no *type-param-value* has been specified is its default value.

7.5.10 Construction of derived-type values

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.

R756 *structure-constructor* is *derived-type-spec* ([*component-spec-list*])

R757 *component-spec* is [*keyword* =] *component-data-source*

R758 *component-data-source* is *expr*
or *data-target*
or *proc-target*

C7100 (R756) The *derived-type-spec* shall not specify an *abstract type* (7.5.7).

C7101 (R756) At most one *component-spec* shall be provided for a component.

C7102 (R756) If a *component-spec* is provided for an ancestor component, a *component-spec* shall not be provided for any component that is *inheritance associated* with a *subcomponent* of that ancestor component.

C7103 (R756) A *component-spec* shall be provided for a nonallocatable component unless it has *default initialization* or is *inheritance associated* with a *subcomponent* of another component for which a

1 *component-spec* is provided.

2 C7104 (R757) The *keyword*= shall not be omitted from a *component-spec* unless the *keyword*= has been
3 omitted from each preceding *component-spec* in the constructor.

4 C7105 (R757) Each *keyword* shall be the name of a component of the type.

5 C7106 (R756) The type name and all components of the type for which a *component-spec* appears shall
6 be accessible in the *scoping unit* containing the *structure constructor*.

7 C7107 (R756) If *derived-type-spec* is a type name that is the same as a generic name, the *component-spec-*
8 *list* shall not be a valid *actual-arg-spec-list* for a function reference that is resolvable as a generic
9 reference to that name (15.5.5.2).

10 C7108 (R758) A *data-target* shall correspond to a *data pointer* component; a *proc-target* shall correspond
11 to a *procedure pointer* component.

12 C7109 (R758) If *expr* is a *coindexed object*, it shall not have a pointer component at any level of component
13 selection.

14 C7110 (R758) A *data-target* shall have the same *rank* as its corresponding component.

NOTE 1

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*.

NOTE 2

Although a *coindexed object* with a pointer subcomponent is not the only way for a structure constructor to produce a value with an undefined pointer subcomponent, copying a pointer from another *image* is particularly likely to cause undiagnosed incorrect results, and is thus precluded in this context.

15 2 In the absence of a *component keyword*, each *component-data-source* is assigned to the corresponding
16 *component* in *component order* (7.5.4.7). If a *component keyword* appears, the *expr* is assigned to the
17 *component* named by the keyword. For a nonpointer *component*, the *declared type* and *type parameters*
18 of the *component* and *expr* shall conform in the same way as for a *variable* and *expr* in an *intrinsic assign-*
19 *ment statement* (10.2.1.2). If necessary, each value of intrinsic type is converted according to the rules of
20 intrinsic assignment (10.2.1.3) to a value that agrees in type and type parameters with the corresponding
21 component of the derived type. For a nonpointer nonallocatable component, the shape of the expression
22 shall conform with the shape of the component.

23 3 If a component with *default initialization* has no corresponding *component-data-source*, then the *default*
24 *initialization* is applied to that component. If an *allocatable* component has no corresponding *component-*
25 *data-source*, then that component has an allocation status of unallocated.

NOTE 3

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 7.5.7.2, NOTE 4:

```
! 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)
```


NOTE 3 (cont.)

! need to be accessible.

- 1 4 A **structure constructor** shall not appear before the referenced type is defined.
- 2 5 For a pointer component, the corresponding **component-data-source** shall be an allowable **data-target** or
- 3 **proc-target** for such a pointer in a **pointer assignment statement** (10.2.2). If the component data source
- 4 is a pointer, the association of the component is that of the pointer; otherwise, the component is **pointer**
- 5 **associated** with the component data source.

NOTE 4

For example, if the variable TEXT were declared (8.2) to be

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

and BIBLIO were declared using the derived-type definition REFERENCE in 7.5.4.4, NOTE

```
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 6 If a component of a derived type is **allocatable**, the corresponding constructor expression shall be a ref-
- 7 erence to the intrinsic function **NULL** with no arguments, an **allocatable** entity of the same **rank**, or shall
- 8 evaluate to an entity of the same **rank**. If the expression is a reference to the intrinsic function **NULL**, the
- 9 corresponding component of the constructor has a status of unallocated.
- 10 7 If the component is **allocatable** and the expression is an **allocatable** entity, the corresponding component
- 11 of the constructor has the same allocation status as that **allocatable** entity. If it is allocated, it has the same
- 12 **bounds**; if a length parameter of the component is **deferred**, its value is the same as the corresponding
- 13 parameter of the expression. If the component is **polymorphic**, it has the same **dynamic type** and value;
- 14 otherwise, it has the value converted, if necessary, to the **declared type** of the component.
- 15 8 If the component is **allocatable** and the expression is not an **allocatable** entity, the component has an alloc-
- 16 ation status of allocated and the same **bounds** as the expression; if a length parameter of the component
- 17 is **deferred**, its value is the same as the corresponding parameter of the expression. If the component is
- 18 **polymorphic**, it has the same **dynamic type** and value; otherwise, it has the value converted, if necessary,
- 19 to the **declared type** of the component.

NOTE 5

This example shows a derived-type constant expression using the derived type defined in 7.5.2.1, NOTE:

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

This could also be written as

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

NOTE 6

An example constructor using the derived type GENERAL_POINT defined in 7.5.3.1, NOTE is

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

7.5.11 Derived-type operations and assignment

Intrinsic assignment of derived-type entities is described in 10.2.1. This document does not specify any intrinsic operations on derived-type entities. Any operation on derived-type entities or defined assignment (10.2.1.4) for derived-type entities shall be defined explicitly by a function or a subroutine, and a generic interface (7.5.5, 15.4.3.2).

7.6 Other nonintrinsic types**7.6.1 Interoperable enumerations and enum types**

An interoperable enumeration is a set of interoperable enumerators, optionally together with an enum type. An *enum-def* defines an interoperable enumeration. An interoperable enumerator is a named integer constant; all the enumerators defined by a particular *enum-def* have the same kind. An enum type is a nonintrinsic type that is not a derived type; it has no type parameter.

R759 *enum-def* **is** *enum-def-stmt*
 enumerator-def-stmt
 [*enumerator-def-stmt*] ...
 end-enum-stmt

R760 *enum-def-stmt* **is** ENUM, BIND(C) [:: *enum-type-name*]

R761 *enumerator-def-stmt* **is** ENUMERATOR [::] *enumerator-list*

R762 *enumerator* **is** *named-constant* [= *scalar-int-constant-expr*]
 or *named-constant* [= *boz-literal-constant*]

R763 *end-enum-stmt* **is** END ENUM

C7111 (R761) If = appears in an *enumerator*, a double-colon separator shall appear before the *enumerator-list*.

R764 *enum-type-spec* **is** *enum-type-name*

C7112 An *enum-type-name* in an *enum-type-spec* shall be the name of a previously defined enum type.

The kind type parameter of each enumerator defined by an *enum-def* is the kind that is interoperable (18.3.1) with the corresponding C enumerated type. The corresponding C enumerated type is the type that would be declared by a C enumeration specifier (ISO/IEC 9899:2018, 6.7.2.2) 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*.

If *enum-type-name* appears in an *enum-def*, the *enum-def* defines the enum type with that name. An enum type is an interoperable type. The set of values of an enum type has a one-to-one correspondence with the set of possible values for the integer kind of its enumerators. The internal representation of each enum type value is the same as that of the corresponding integer.

1 4 An enum type specifier specifies the type. Two data entities of enum type have the same type if they are
2 declared with reference to the same enum type definition.

3 5 The [companion processor](#) (5.5.7) shall be one that uses the same representation for the types declared by
4 all C enumeration specifiers that specify the same values in the same order.

NOTE 1

If a [companion processor](#) uses an unsigned type to represent a C enumerated type, the Fortran processor will use the signed integer type of the same width for the enumeration, even though some of the values of the C enumerators might not be representable in this signed integer type. The types of any such enumerators will be [interoperable](#) with the type declared in the C enumeration.

NOTE 2

ISO/IEC 9899:2018 guarantees the enumeration constants fit in a C int (ISO/IEC 9899:2018, 6.7.2.2). 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 3

ISO/IEC 9899:2018 specifies that two C enumerated types are compatible only if they specify enumeration constants with the same names and same values in the same order. This document further requires that a C processor that is to be a [companion processor](#) of a Fortran processor use the same representation for two C enumerated types if they both specify enumeration constants with the same values in the same order, even if the names are different.

5 6 An enumerator is treated as if it were explicitly declared with the [PARAMETER attribute](#). The enumerator
6 is a scalar [named constant](#), with the value determined as follows.

- 7 • If *scalar-int-constant-expr* appears, the enumerator has the value specified by *scalar-int-constant-*
8 *expr*.
- 9 • If *boz-literal-constant* appears, the enumerator has the value specified by INT (*boz-literal-constant*,
10 C_INT), where C_INT is from the intrinsic module ISO_C_BINDING.
- 11 • If neither *scalar-int-constant-expr* nor *boz-literal-constant* appears and the enumerator is the first
12 enumerator in *enum-def*, the enumerator has the value zero.
- 13 • If neither *scalar-int-constant-expr* nor *boz-literal-constant* appears and the enumerator is not the first
14 enumerator in *enum-def*, it has the value obtained by adding one to the value of the enumerator that
15 immediately precedes it in the *enum-def*.

16 R765 *enum-constructor* is *enum-type-spec* (*scalar-expr*)

17 C7113 The *scalar-expr* in an *enum-constructor* shall be of type integer or be a *boz-literal-constant*.

18 7 An enum constructor produces a scalar value of the specified type, with the specified internal representa-
19 tion. The value of *scalar-expr* shall be representable in objects of that type.

NOTE 4

Example of an interoperable enumeration definition:

```
ENUM, BIND(C)
  ENUMERATOR :: RED = 4, BLUE = 9
  ENUMERATOR YELLOW
END ENUM
```

The [kind type parameter](#) for this enumeration is processor dependent, but the processor is required to select a kind sufficient to represent the values 4, 9, and 10, which are the values of its enumerators. The following declaration might be equivalent to the above enumeration definition.

NOTE 4 (cont.)

```
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 5

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.

NOTE 6

Here is an example of a module that defines two enum types.

```
Module enum_mod
  Enum, Bind(C) :: myenum
  Enumerator :: one=1, two, three
End Enum
Enum, Bind(C) :: flags
  Enumerator :: f1 = 1, f2 = 2, f3 = 4
End Enum
Contains
  Subroutine sub(a) Bind(C)
    Type(myenum), Value :: a
    Print *,a ! Prints the integer value, as if it were Print *,Int(a).
  End Subroutine
End Module
```

Here is a simple program that uses that module and the enum constructor.

```
Program example
  Use enum_mod
  Type(myenum) :: x = one           ! Assign enumerator to enum-type var.
  Type(myenum) :: y = myenum(12345) ! Using the constructor.
  Type(myenum) :: x2 = myenum(two) ! Constructor not needed but valid.
  Call sub(x)
  Call sub(three)
  Call sub(myenum(-Huge(one)))
End Program
```

Here is an example of invalid usage.

```
Program invalid
  Use enum_mod
  Type(myenum) :: z = 12345         ! Integer expr with no enumerator.
  Call sub(999)                     ! Not type-compatible (constructor needed).
  Call sub(f1)                      ! Wrong enum type.
End Program
```

1 7.6.2 Enumeration types

- 2 1 An enumeration type is a nonintrinsic type with no type parameter. It is not a derived type and is not
3 interoperable. An enumeration type definition defines the name of the type and lists all the possible values
4 of the type.

5 R766 *enumeration-type-def* is *enumeration-type-stmt*
6 *enumeration-enumerator-stmt*
7 [*enumeration-enumerator-stmt*]...
8 *end-enumeration-type-stmt*

- 1 R767 *enumeration-type-stmt* **is** ENUMERATION TYPE [[, *access-spec*] ::] *enumeration-type-name*
- 2 C7114 An *access-spec* on an *enumeration-type-stmt* shall only appear in the specification part of a module.
- 3 R769 *enumeration-enumerator-stmt* **is** ENUMERATOR [::] *enumerator-name-list*
- 4 R770 *end-enumeration-type-stmt* **is** END ENUMERATION TYPE [*enumeration-type-name*]
- 5 C7115 If *enumeration-type-name* appears on an END ENUMERATION TYPE statement, it shall be the same
- 6 as on the ENUMERATION TYPE statement.
- 7 2 The *access-spec* on an ENUMERATION TYPE statement specifies the accessibility of the *enumeration-type-*
- 8 *name* and the default accessibility of its enumerators. The accessibility of an enumerator may be confirmed
- 9 or overridden by an *access-stmt*.
- 10 3 Each enumerator in the definition is a scalar *named constant* of the enumeration type. The order of the
- 11 enumerator names in the definition defines the ordinal position of each enumerator.
- 12 R771 *enumeration-type-spec* **is** *enumeration-type-name*
- 13 C7116 The *enumeration-type-name* in an *enumeration-type-spec* shall be the name of a previously defined
- 14 enumeration type.
- 15 4 An enumeration type specifier specifies the type. Two data entities of enumeration type have the same
- 16 type if they are declared with reference to the same enumeration type definition.
- 17 R772 *enumeration-constructor* **is** *enumeration-type-spec* (*scalar-int-expr*)
- 18 5 An enumeration constructor produces the scalar value of the enumeration type whose ordinal position is
- 19 the value of the *scalar-int-expr*. The *scalar-int-expr* shall have a value that is positive and less than or equal
- 20 to the number of enumerators in the enumeration type's definition.

NOTE

Here is an example of a module defining two enumeration types.

```

Module enumeration_mod
  Enumeration Type :: v_value
    Enumerator :: v_one, v_two, v_three
    Enumerator v_four
  End Enumeration Type
  Enumeration Type :: w_value
    Enumerator :: w1, w2, w3, w4, w5, wendsentinel
  End Enumeration Type
Contains
  Subroutine sub(a)
    Type(v_value), Intent(In) :: a
    Print I, a ! Acts similarly to Print *, Int(a).
1   Format('A has ordinal value ', I0)
  End Subroutine
  Subroutine wcheck(w)
    Type(w_value), Intent(In) :: w
    Select Case(w)
    Case(w1)
      Print *, 'w1 selected'
    Case (w2:w4)
      Print *, 'One of w2...w4 selected'
    Case (wendsentinel)
      Stop 'Invalid w selected'
    Case Default
      Stop 'Unrecognized w selected'
    End Select
  End Subroutine

```

NOTE (cont.)

End Module

Here is an example of a program using that module.

```

Program example
  Use enumeration_mod
  Type(v_value) :: x = v_one
  Type(v_value) :: y = v_value(2)    ! Explicit constructor producing v_two.
  Type(v_value) :: z,nz              ! Initially undefined.
  Call sub(x)
  Call sub(v_three)
  z = v_value(1)                     ! First value.
  Do
    If (z==Huge(x)) Write (*,'(A)',Advance='No') ' Huge:'
    Call sub(z)
    nz = Next(z)
    If (z==nz) Exit
    z = nz
  End Do
End Program

```

Here is an example showing some invalid usages of enumerations.

```

Program invalid
  Use enumeration_mod
  Type(v_value) :: a, b
  a = 1          ! INVALID - wrong type (INTEGER).
  b = w1         ! INVALID - wrong enumeration type.
  Print *,a      ! INVALID - list-directed i/o not available.
End Program

```

An enumeration type can be used to declare components, for example:

```

Module example2
  Use enumeration_mod
  Type vw
    Type(v_value) v
    Type(w_value) w
  End Type
  Contains
    Subroutine showme(ka)
      Type(vw),Intent(In) :: ka
      Print 1,ka
1     Format(1X,'v ordinal is ',I0,', w ordinal is ',I0)
    End Subroutine
End Module

```

7.7 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.

R773 *boz-literal-constant* **is** *binary-constant*
 or *octal-constant*
 or *hex-constant*

R774 *binary-constant* **is** B ' digit [digit] ... '
 or B " digit [digit] ... "

C7117 (R774) *digit* shall have one of the values 0 or 1.

R775 *octal-constant* **is** O ' digit [digit] ... '

1 **or** 0 " *digit* [*digit*] ... "

2 C7118 (R775) *digit* shall have one of the values 0 through 7.

3 R776 *hex-constant* **is** Z ' *hex-digit* [*hex-digit*] ... '

4 **or** Z " *hex-digit* [*hex-digit*] ... "

5 R777 *hex-digit* **is** *digit*

6 **or** A

7 **or** B

8 **or** C

9 **or** D

10 **or** E

11 **or** F

- 12 2 The *hex-digits* A through F represent the numbers ten through fifteen, respectively; they may be represented by their lower-case equivalents. Each digit of a *boz-literal-constant* represents a sequence of bits, according to its numerical interpretation, using the model of 16.3, with z equal to one for binary constants, three for octal constants or four for hexadecimal constants. A *boz-literal-constant* represents a sequence of bits that consists of the concatenation of the sequences of bits represented by its digits, in the order the digits are specified. The positions of bits in the sequence are numbered from right to left, with the position of the rightmost bit being zero. The length of a sequence of bits is the number of bits in the sequence. The processor shall allow the position of the leftmost nonzero bit to be at least $z - 1$, where z is the maximum value that could result from invoking the intrinsic function *STORAGE_SIZE* (16.9.200) with an argument that is a real or integer scalar of any kind supported by the processor.

22 C7119 (R773) A *boz-literal-constant* shall appear only as a *data-stmt-constant* in a *DATA statement*, as the *initialization* for a *named constant* or variable of type integer or real, as the value in an *enumerator*, as the *expr* in an intrinsic assignment whose *variable* is of type integer or real, as an *ac-value* in an array constructor with a *type-spec* that specifies type integer or real, as the *scalar-expr* in an enum constructor, or where explicitly allowed in 16.9 as an actual argument of an *intrinsic* procedure.

27 7.8 Construction of array values

- 28 1 An array constructor constructs a rank-one array value from a sequence of scalar values, array values, and
29 implied DO loops.

30 R778 *array-constructor* **is** (/ *ac-spec* /)

31 **or** *lbracket ac-spec rbracket*

32 R779 *ac-spec* **is** *type-spec* ::

33 **or** [*type-spec* ::] *ac-value-list*

34 R780 *lbracket* **is** [

35 R781 *rbracket* **is**]

36 R782 *ac-value* **is** *expr*

37 **or** *ac-implied-do*

38 R783 *ac-implied-do* **is** (*ac-value-list* , *ac-implied-do-control*)

- 1 R784 *ac-implied-do-control* is [*integer-type-spec* ::] *ac-do-variable* = *scalar-int-expr* , ■
 2 ■ *scalar-int-expr* [, *scalar-int-expr*]
- 3 R785 *ac-do-variable* is *do-variable*
- 4 C7120 (R779) If *type-spec* is omitted, each *ac-value* expression in the *array-constructor* shall have the
 5 same *declared type* and *kind type parameters*.
- 6 C7121 (R779) If *type-spec* specifies an intrinsic type or enum type, each *ac-value* expression in the *array-*
 7 *constructor* shall be of a type that is in type conformance with a variable of type *type-spec* as spe-
 8 cified in Table 10.8, or be a *boz-literal-constant*.
- 9 C7122 (R779) If *type-spec* specifies a *derived type*, the *declared type* of each *ac-value* expression in the
 10 *array-constructor* shall be that *derived type* and shall have the same *kind type parameter* values as
 11 specified by *type-spec*.
- 12 C7123 (R779) If *type-spec* specifies an enumeration type, each *ac-value* shall be of that type.
- 13 C7124 (R782) An *ac-value* shall not be *unlimited polymorphic*.
- 14 C7125 (R782) The *declared type* of an *ac-value* shall not be abstract.
- 15 C7126 If an *ac-value* is a *boz-literal-constant*, *type-spec* shall appear and shall specify type integer or real.
- 16 C7127 If an *ac-value* is a *boz-literal-constant* and *type-spec* specifies type real, the *boz-literal-constant* shall
 17 be a valid internal representation for the specified kind of real.
- 18 C7128 (R783) The *ac-do-variable* of an *ac-implied-do* that is in another *ac-implied-do* shall not appear as
 19 the *ac-do-variable* of the containing *ac-implied-do*.
- 20 2 If *type-spec* is omitted, corresponding *length type parameters* of the *declared type* of each *ac-value* expres-
 21 sion shall have the same value; in this case, the *declared type* and *type parameters* of the array constructor
 22 are those of the *ac-value* expressions.
- 23 3 If *type-spec* appears, it specifies the *declared type* and type parameters of the array constructor. Each *ac-*
 24 *value* expression in the *array-constructor* shall be compatible with intrinsic assignment to a variable of this
 25 type and type parameters. Each value is converted to the type and *type parameters* of the *array-constructor*
 26 in accordance with the rules of intrinsic assignment (10.2.1.3).
- 27 4 The *dynamic type* of an array constructor is the same as its *declared type*.
- 28 5 The character length of an *ac-value* in an *ac-implied-do* whose iteration count is zero shall not depend on
 29 the value of the *ac-do-variable* and shall not depend on the value of an expression that is not a *constant*
 30 *expression*.
- 31 6 If an *ac-value* is a scalar expression, its value specifies an element of the array constructor. If an *ac-value* is
 32 an array expression, the values of the elements of the expression, in array element order (9.5.3.3), specify
 33 the corresponding sequence of elements of the array constructor. If an *ac-value* is an *ac-implied-do*, it is
 34 expanded to form a sequence of elements under the control of the *ac-do-variable*, as in the *DO construct*
 35 (11.1.7.4).
- 36 7 For an *ac-implied-do*, the loop initialization and execution is the same as for a *DO construct*. The scope and
 37 attributes of an *ac-do-variable* are described in 19.4.

- 1 8 An empty sequence forms a zero-sized array.

NOTE 1

A one-dimensional array can be reshaped into any allowable array shape using the intrinsic function [RESHAPE \(16.9.175\)](#). 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 2

Examples of array constructors containing an implied DO are:

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

and

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

NOTE 3

Using the type definition for PERSON in [7.5.2.1, NOTE](#), an example of the construction of a derived-type array value is:

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

NOTE 4

Using the type definition for LINE in [7.5.4.2, NOTE 1](#), an example of the construction of a derived-type scalar value with a rank-two 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 5

Examples of zero-size array constructors are:

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

NOTE 6

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.

8 Attribute declarations and specifications

8.1 Attributes of procedures and data objects

- 1 Every data object has a type and **rank** and can have **type parameters** and other properties that determine the uses of the object. Collectively, these properties are the **attributes** of the object. The **declared type** of a named data object is either specified explicitly in a **type declaration statement** or determined implicitly by the first letter of its name (8.7). The **attributes** listed in 8.5 can be specified in a **type declaration statement** or individually in separate specification statements.
- 2 A function has a type and **rank** and can 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 can have other **attributes** that determine the uses of the subroutine.

8.2 Type declaration statement

R801 *type-declaration-stmt* **is** *declaration-type-spec* [[, *attr-spec*] ... ::] *entity-decl-list*

- 1 The type declaration statement specifies the **declared type** of the entities in the entity declaration list. The type and type parameters are those specified by *declaration-type-spec*, except that the character length type parameter can be overridden for an entity by the appearance of * *char-length* in its *entity-decl*.

R802 *attr-spec* **is** *access-spec*
 or ALLOCATABLE
 or ASYNCHRONOUS
 or CODIMENSION *lbracket coarray-spec rbracket*
 or CONTIGUOUS
 or DIMENSION (*array-spec*)
 or EXTERNAL
 or INTENT (*intent-spec*)
 or INTRINSIC
 or *language-binding-spec*
 or OPTIONAL
 or PARAMETER
 or POINTER
 or PROTECTED
 or *rank-clause*
 or SAVE
 or TARGET
 or VALUE
 or VOLATILE

C801 (R801) The same *attr-spec* shall not appear more than once in a given *type-declaration-stmt*.

1	C802	(R801) If a <i>language-binding-spec</i> with a NAME= specifier appears, the <i>entity-decl-list</i> shall consist
2		of a single <i>entity-decl</i> .
3	C803	(R801) If a <i>language-binding-spec</i> is specified, the <i>entity-decl-list</i> shall not contain any procedure
4		names.
5	2	The type declaration statement also specifies the <i>attributes</i> whose keywords appear in the <i>attr-spec</i> , except
6		that the <i>DIMENSION attribute</i> can be specified or overridden for an entity by the appearance of <i>array-</i>
7		<i>spec</i> in its <i>entity-decl</i> , and the <i>CODIMENSION attribute</i> can be specified or overridden for an entity by the
8		appearance of <i>coarray-spec</i> in its <i>entity-decl</i> .
9	R803	<i>entity-decl</i> is <i>object-name</i> [(<i>array-spec</i>)] ■
10		■ [<i>lbracket coarray-spec rbracket</i>] ■
11		■ [* <i>char-length</i>] [<i>initialization</i>]
12		or <i>function-name</i> [* <i>char-length</i>]
13	C804	(R803) If the entity is not of type character, * <i>char-length</i> shall not appear.
14	C805	A <i>type-param-value</i> in a <i>char-length</i> in an <i>entity-decl</i> shall be a colon, asterisk, or specification ex-
15		pression.
16	C806	(R801) If <i>initialization</i> appears, a double-colon separator shall appear before the <i>entity-decl-list</i> .
17	C807	(R801) If the PARAMETER keyword appears, <i>initialization</i> shall appear in each <i>entity-decl</i> .
18	C808	(R803) An <i>initialization</i> shall not appear if <i>object-name</i> is a <i>dummy argument</i> , a function result, an
19		object in a named <i>common block</i> ¹ unless the type declaration is in a block data program unit, an
20		object in <i>blank common</i> , an <i>allocatable</i> variable, or an <i>automatic data object</i> .
21	C809	(R803) The <i>function-name</i> shall be the name of an external function, an intrinsic function, a <i>dummy</i>
22		<i>function</i> , a <i>procedure pointer</i> , or a statement function ² .
23	R804	<i>object-name</i> is <i>name</i>
24	C810	(R804) The <i>object-name</i> shall be the name of a data object.
25	R805	<i>initialization</i> is = <i>constant-expr</i>
26		or => <i>null-init</i>
27		or => <i>initial-data-target</i>
28	R806	<i>null-init</i> is <i>function-reference</i>
29	C811	(R803) If => appears in <i>initialization</i> , the entity shall have the <i>POINTER attribute</i> . If = appears in
30		<i>initialization</i> , the entity shall not have the <i>POINTER attribute</i> .
31	C812	(R803) If <i>initial-data-target</i> appears, <i>object-name</i> shall be data-pointer-initialization compatible
32		with it (7.5.4.6).
33	C813	(R806) The <i>function-reference</i> shall be a reference to the intrinsic function <i>NULL</i> with no argu-
34		ments.
35	3	A name that identifies a specific intrinsic function ³ has a type as specified in 16.8. An explicit type declar-

¹Common blocks are obsolescent.

²Statement functions are obsolescent.

³Specific intrinsic function names are obsolescent.

ation statement is not required; however, it is permitted. Specifying a type for a generic intrinsic function name in a type declaration statement has no effect.

4 If *initialization* appears for a nonpointer entity,

- its type and *type parameters* shall conform as specified for intrinsic assignment (10.2.1.2);
- if the entity has implied shape, the rank of *initialization* shall be the same as the rank of the entity;
- if the entity does not have implied shape, *initialization* shall either be scalar or have the same shape as the entity.

NOTE

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.
TYPEOF (K) K TMP      ! Also has range at least -9999 to 9999.
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
CLASSOF (MAT), POINTER :: MAT_REF ! Same declared type and type parameters as MAT.
```

(The type HUMONGOUS_MATRIX is defined in 7.5.3.1, NOTE.)

8.3 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*.

C814 An *automatic data object* shall not have the *SAVE attribute*.

2 If a type parameter in a *declaration-type-spec* or in a *char-length* in an *entity-decl* for a *local variable* of a subprogram or *BLOCK construct* is defined by an expression that is not a *constant expression*, the type parameter value is established on entry to a procedure defined by the subprogram, or on execution of the *BLOCK statement*, and is not affected by any redefinition or undefinition of the variables in the expression during execution of the procedure or *BLOCK construct*.

8.4 Initialization

1 The appearance of *initialization* in an *entity-decl* for an entity without the *PARAMETER attribute* specifies that the entity is a variable with *explicit initialization*. *Explicit initialization* alternatively may be specified in a *DATA statement* unless the variable is of a derived type for which *default initialization* is specified. If *initialization* = *constant-expr*, the variable is initially defined with the value specified by the *constant-expr*; if necessary, the value is converted according to the rules of intrinsic assignment (10.2.1.3) to a value that agrees in type, type parameters, and shape with the variable. A variable, or part of a variable, shall not be *explicitly initialized* more than once in a program. If the variable is an array, it shall have its shape specified in either the type declaration statement or a previous attribute specification statement in the same *scoping unit*.

1 2 If *null-init* appears, the initial association status of the object is *disassociated*. If *initial-data-target* appears,
2 the object is initially associated with the *target*.

3 3 **Explicit initialization** of a variable that is not in a **common block**⁴ implies the **SAVE attribute**, which may be
4 confirmed by explicit specification.

5 8.5 Attributes

6 8.5.1 Attribute specification

7 1 An **attribute** may be explicitly specified by an *attr-spec* in a **type declaration statement** or by an attribute
8 specification statement (8.6). The following constraints apply to **attributes**.

9 C815 An entity shall not be explicitly given any **attribute** more than once in a **scoping unit**.

10 C816 An *array-spec* for a nonallocatable nonpointer function result shall be an *explicit-shape-spec-list*.

11 8.5.2 Accessibility attribute

12 1 The accessibility attribute specifies the accessibility of an entity via a particular identifier.

```

13      R807  access-spec      is PUBLIC
14                                     or PRIVATE

```

15 C817 An *access-spec* shall appear only in the *specification-part* of a module.

2 An *access-spec* in a type declaration statement specifies the accessibility of the names of all the entities
declared by that statement. An *access-spec* in a *derived-type-stmt* specifies the accessibility of the derived
type name. An *access-spec* in an *enumeration-type-stmt* specifies the accessibility of the enumeration type
name, and the default accessibility of its enumerators. Accessibility can also be specified by an *access-stmt*.

20 3 An identifier that is specified in a module or is accessible in a module by use association has either the **PUB-**
21 **LIC attribute** or **PRIVATE attribute**. An identifier whose accessibility is not explicitly specified has default
22 accessibility (8.6.1).

23 4 The default accessibility attribute for a module is PUBLIC unless it has been changed by a PRIVATE state-
24 ment. Only an identifier that has the PUBLIC attribute in that module is available to be accessed from that
25 module by use association.

NOTE 1

An identifier can only be accessed by [use association](#) if it has the [PUBLIC attribute](#) in the module from which it is accessed. It can nonetheless have the [PRIVATE attribute](#) in a module in which it is accessed by [use association](#), and therefore not be available by [use association](#) from that module.

NOTE 2

An example of an accessibility specification is:

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

26 8.5.3 ALLOCATABLE attribute

27 1 A variable with the **ALLOCATABLE** attribute is a variable for which space is allocated during execution.

⁴Common blocks are obsolescent.

NOTE

Only variables and components can have the ALLOCATABLE attribute. The result of referencing a function whose result variable has the ALLOCATABLE attribute is a value that does not itself have the ALLOCATABLE attribute.

8.5.4 ASYNCHRONOUS attribute

- 1 An entity with the **ASYNCHRONOUS attribute** is a variable, and may be subject to asynchronous input/output or asynchronous communication.
- 2 The **base object** of a variable shall have the **ASYNCHRONOUS attribute** in a **scoping unit** if
 - the variable is a **dummy argument** or appears in an executable statement or **specification expression** in that **scoping unit**, and
 - any statement of the **scoping unit** is executed while the variable is a pending input/output storage sequence affector (12.6.2.5) or a pending communication affector (18.10.4).
- 3 Use of a variable in an asynchronous data transfer statement can imply the **ASYNCHRONOUS attribute**; see 12.6.2.5.
- 4 An object with the **ASYNCHRONOUS attribute** may be associated with an object that does not have the **ASYNCHRONOUS attribute**, including by **use** (14.2.2) or **host** association (19.5.1.4). If an object that is not a **local variable** of a **BLOCK construct** is specified to have the **ASYNCHRONOUS attribute** in the **specification-part** of the construct, the object has the attribute within the construct even if it does not have the attribute outside the construct. If an object has the **ASYNCHRONOUS attribute**, then all of its subobjects also have the **ASYNCHRONOUS attribute**.

NOTE

The **ASYNCHRONOUS attribute** specifies the variables that might be associated with a pending input/output storage sequence (the actual memory locations on which asynchronous input/output is being performed) while the **scoping unit** is in execution. This information could be used by the compiler to disable certain code motion optimizations.

8.5.5 BIND attribute for data entities

- 1 The **BIND attribute** for a variable or **common block** specifies that it is capable of interoperating with a C variable whose name has external linkage (18.9). Common blocks, with or without the **BIND attribute**, are obsolescent. The **BIND attribute** for a **common block** implies the **SAVE attribute**, which may be confirmed by explicit specification.

R808 *language-binding-spec* **is** BIND (C [, NAME = *scalar-default-char-constant-expr*])

C818 An entity with the **BIND attribute** shall be a **common block**, variable, type, or procedure.

C819 A variable with the **BIND attribute** shall be declared in the specification part of a module.

C820 A variable with the **BIND attribute** shall be **interoperable** (18.3).

C821 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

ISO/IEC 9899:2018 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 (ISO/IEC 9899:2018, 6.4.3). 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.

8.5.6 CODIMENSION attribute**8.5.6.1 General**

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

R809 **coarray-spec** is *deferred-coshape-spec-list*
or *explicit-coshape-spec*

C822 The sum of the **rank** and **corank** of an entity shall not exceed fifteen.

C823 A **coarray** shall be a component or a variable that is not a function result.

C824 A **coarray** shall not be of type **C_PTR** or type **C_FUNPTR** from the intrinsic module **ISO_C_BINDING** (18.3.2), or of type **TEAM_TYPE** from the intrinsic module **ISO_FORTRAN_ENV** (16.10.2.34).

C825 An entity whose type has a **coarray potential subobject component** shall not be a pointer, shall not be a **coarray**, and shall not be a function result.

C826 A **coarray** or an object with a **coarray potential subobject component** shall be an **associate name** or a **dummy argument**, or have the **ALLOCATABLE** or **SAVE** attribute.

NOTE 1

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 2

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 3

Examples of **CODIMENSION attribute** specifications are:

```
REAL W(100,100) [0:2,*] ! Explicit-shape coarray
REAL, CODIMENSION[*] :: X ! Scalar coarray
REAL, CODIMENSION[3,*] :: Y(:) ! Assumed-shape coarray
REAL, CODIMENSION[:,ALLOCATABLE] :: Z(:, :) ! Allocatable coarray
```


8.5.6.2 Allocatable coarray

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

R810 *deferred-coshape-spec* is :

C827 A **coarray** with the **ALLOCATABLE** attribute shall have a *coarray-spec* that is a *deferred-coshape-spec-list*.

2 The **corank** of an **allocatable coarray** is equal to the number of colons in its *deferred-coshape-spec-list*.

3 The **cobounds** of an unallocated **allocatable coarray** are undefined. No part of such a **coarray** shall be referenced or defined; however, the **coarray** may appear as an argument to an intrinsic **inquiry function** as specified in 16.1.

4 The **cobounds** of an allocated **allocatable coarray** are those specified when the **coarray** is allocated.

5 The **cobounds** of an **allocatable coarray** are unaffected by any subsequent redefinition or undefinition of the variables on which the **cobounds**' expressions depend.

8.5.6.3 Explicit-coshape coarray

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

R811 *explicit-coshape-spec* is [[*lower-cobound* :] *upper-cobound*,]... ■
■ [*lower-cobound* :] *

C828 A nonallocatable **coarray** shall have a *coarray-spec* that is an *explicit-coshape-spec*.

2 The **corank** is equal to one plus the number of *upper-cobounds*.

R812 *lower-cobound* is *specification-expr*

R813 *upper-cobound* is *specification-expr*

C829 (R811) 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**.

8.5.7 CONTIGUOUS attribute

C830 An entity with the **CONTIGUOUS** attribute shall be an **array pointer**, an **assumed-shape array**, or an **assumed-rank dummy data object**.

- 1 The **CONTIGUOUS attribute** specifies that an **assumed-shape array** is contiguous, that an **array pointer** can only be **pointer associated** with a **contiguous target**, or that an **assumed-rank dummy data object** is contiguous.
- 2 An object is **contiguous** if it is
- (1) an object with the **CONTIGUOUS attribute**,
 - (2) a nonpointer whole array that is not **assumed-shape**,
 - (3) an **assumed-shape array** that is argument associated with an array that is contiguous,
 - (4) an **assumed-rank dummy data object** whose **effective argument** is contiguous,
 - (5) an array allocated by an **ALLOCATE statement**,
 - (6) a **pointer associated** with a contiguous **target**, or
 - (7) a nonzero-sized **array section** (9.5.3) provided that
 - (a) its **base object** is **contiguous**,
 - (b) it does not have a **vector subscript**,
 - (c) the array element ordering of the elements of the section is the same as the array element ordering of those elements of the **base object**,
 - (d) in the array element ordering of the **base object**, every element of the **base object** that is not an element of the section either precedes every element of the section or follows every element of the section,
 - (e) if the array is of type character and a **substring-range** appears, the **substring-range** specifies all of the characters of the **parent-string** (9.4.1),
 - (f) only its final **part-ref** has nonzero **rank**, and
 - (g) it is not the real or imaginary part (9.4.4) of an array of type complex.
- 3 An object is not contiguous if it is an array subobject, and
- the object has two or more elements,
 - the elements of the object in array element order are not consecutive in the elements of the **base object**,
 - the object is not of type character with length zero, and
 - the object is not of a derived type that has no **ultimate components** other than zero-sized arrays and characters with length zero.
- 4 It is processor dependent whether any other object is contiguous.

NOTE 1

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 2

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
```

NOTE 3

If an [assumed-shape](#) or [assumed-rank dummy argument](#) has the [CONTIGUOUS attribute](#), there is no requirement for the [actual argument](#) to be contiguous. This is the same as for [dummy arguments](#) that have [explicit shape](#) or [assumed size](#). The [dummy argument](#) will be contiguous even when the [actual argument](#) is not.

8.5.8 DIMENSION attribute

8.5.8.1 General

- 1 The [DIMENSION attribute](#) specifies that an entity is scalar, [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 [RANK](#) clause or its [array-spec](#).

R814 *array-spec* is [explicit-shape-spec-list](#)
or [explicit-shape-bounds-spec](#)
or [assumed-shape-spec-list](#)
or [assumed-shape-bounds-spec](#)
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 1

The maximum [rank](#) of an entity is fifteen minus the [corank](#).

NOTE 2

Examples of [DIMENSION attribute](#) specifications are:

```
SUBROUTINE EX (N, A, B)
  REAL, DIMENSION (N, 10) :: W           ! Automatic explicit-shape array
  REAL, DIMENSION (SHAPE (W)) :: X       ! Array with the same shape as W
  REAL, DIMENSION ([1, 2, 3] : 10) :: Y   ! Same as DIMENSION (1:10, 2:10, 3:10)
  REAL, DIMENSION (LBARRAY:UBARRAY) :: Z ! Upper/lower bounds provided by arrays
  REAL :: ZZ (LBARRAY+2:UBARRAY+2)       ! Upper/lower bounds provided by arrays
  REAL A (:), B (0:)                     ! Assumed-shape arrays
  REAL C (LBARRAY:)                       ! Specified lower bounds, assumed shape
  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
```

8.5.8.2 Explicit-shape array

R815 *explicit-shape-spec* is [\[lower-bound : \] upper-bound](#)

R816 *lower-bound* is [specification-expr](#)

R817 *upper-bound* is [specification-expr](#)

R818 *explicit-shape-bounds-spec* is [\[explicit-bounds-expr : \] explicit-bounds-expr](#)
or [lower-bound : explicit-bounds-expr](#)
or [explicit-bounds-expr : upper-bound](#)

R819 *explicit-bounds-expr* is [int-expr](#)

- 1 C831 An *explicit-shape-spec* or *explicit-shape-bounds-spec* whose *bounds* are not *constant expressions*
 2 shall appear only in a subprogram, derived type definition, *BLOCK construct*, or *interface body*.
- 3 C832 If an *explicit-shape-bounds-spec* has two *explicit-bounds-exprs*, they shall have the same size.
- 4 C833 An *explicit-bounds-expr* shall be a restricted expression that is a rank one integer array with con-
 5 stant size.
- 6 1 The rank of an entity declared with an *explicit-shape-spec-list* is equal to the number of *explicit-shape-specs*;
 7 the rank of an entity declared with an *explicit-shape-bounds-spec* is equal to the size of one of the *explicit-*
 8 *bounds-exprs*. If the rank of such an entity is nonzero, the entity is an *explicit-shape array*; otherwise, it is
 9 scalar.
- 10 2 The values of each *lower-bound* and *upper-bound* in an *explicit-shape-spec* determine the bounds along a
 11 particular dimension and hence the extent in that dimension. If *lower-bound* is omitted, the lower bound
 12 is equal to one.
- 13 3 An *explicit-bounds-expr* that appears immediately before a colon specifies the lower bounds; otherwise, it
 14 specifies the upper bounds. The first element specifies the bound for the first dimension, and so on. A
 15 *lower-bound* or *upper-bound* in an *explicit-shape-bounds-spec* specifies the bound for every dimension of
 16 the entity. If no lower bound is specified in an *explicit-shape-bounds-spec*, all the lower bounds are equal to
 17 one.
- 18 4 The value of a lower bound or an upper bound may be positive, negative, or zero. The subscript range of
 19 the array in that dimension is the set of integer values between and including the lower and upper bounds,
 20 provided the upper bound is not less than the lower bound. If the upper bound is less than the lower bound,
 21 the range is empty, the extent in that dimension is zero, and the array is of zero size.
- 22 5 An *explicit-shape array* that is a named *local variable* of a subprogram or *BLOCK construct* may have *bounds*
 23 that are not *constant expressions*. The *bounds*, and hence shape, are determined on entry to a procedure
 24 defined by the subprogram, or on execution of the *BLOCK statement*, by evaluating the bounds' expres-
 25 sions. The *bounds* of such an array are unaffected by the redefinition or undefinition of any variable during
 26 execution of the procedure or *BLOCK construct*.

27 8.5.8.3 Assumed-shape array

- 28 1 An *assumed-shape array* is a nonallocatable nonpointer *dummy argument* array that takes its shape from
 29 its *effective argument*.
- 30 R820 *assumed-shape-spec* is [*lower-bound*] :
- 31 R821 *assumed-shape-bounds-spec* is *explicit-bounds-expr* :
- 32 2 If the *rank* is not specified by a *rank-clause*, it is equal to the number of colons in the *assumed-shape-spec-*
 33 *list*, or the size of the *explicit-bounds-expr* in the *assumed-shape-bounds-spec*. If the rank is nonzero, the
 34 entity is an *assumed-shape array*; otherwise, it is scalar.
- 35 3 If *explicit-bounds-expr* appears it specifies the lower bounds for every dimension; otherwise, if *lower-bound*
 36 appears it specifies the lower bound for that dimension; otherwise the lower bound is equal to one.
- 37 4 The extent of a dimension of an *assumed-shape array dummy argument* is the extent of the corresponding
 38 dimension of its *effective argument*. If the lower bound value is d and the extent of the corresponding
 39 dimension of its *effective argument* is s , then the value of the upper bound is $s + d - 1$.

8.5.8.4 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.)

R822 *deferred-shape-spec* **is** :

C834 An array with the **POINTER** or **ALLOCATABLE** attribute shall be declared with a *rank-clause* or have an *array-spec* that is a *deferred-shape-spec-list*.

2 If the **rank** is not specified by a *rank-clause*, it 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 16.1.

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** (9.7.1) when the **target** is allocated,
- by pointer assignment (10.2.2), or
- if it is a **dummy argument**, by **argument association** with a nonpointer **actual argument** or an associated pointer **effective argument**.

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

8.5.8.5 Assumed-size array

1 An **assumed-size array** is a **dummy argument** array whose size is assumed from that of its **effective argument**, or the associate name of a **RANK (*)** block in a **SELECT RANK construct**. The **rank** and extents may differ for the **effective** and **dummy** arguments; only the size of the **effective argument** is assumed by the **dummy argument**. A **dummy argument** is declared to be an **assumed-size array** by an *assumed-size-spec* or an *implied-shape-or-assumed-size-spec*.

R823 *assumed-implied-spec* **is** [*lower-bound* :] *

R824 *assumed-size-spec* **is** *explicit-shape-spec-list*, *assumed-implied-spec*

C835 An object whose array bounds are specified by an *assumed-size-spec* shall be a **dummy data object**.

C836 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.

R826 *implied-shape-or-assumed-size-spec* **is** *assumed-implied-spec*

C837 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 (9.5.3.3) in an array of size x , the size of the dummy array is $x - r + 1$.
- If the **actual argument** is a default character array, default character array element, or a default character array element substring (9.4.1), and if it begins at **character storage unit** t of an array with c **character storage units**, the size of the dummy array is $\text{MAX}(\text{INT}((c - t + 1)/e), 0)$, 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 $\text{MAX}(\text{INT}(l/e), 0)$, 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-specs*.

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-specs* appears, it specifies the bounds of the first **rank**—1 dimensions. If *lower-bound* appears it specifies the lower bound of the last dimension; otherwise that lower bound is 1. An **assumed-size array** can be subscripted or sectioned (9.5.3).

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.

8.5.8.6 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*.

R827 *implied-shape-spec* **is** *assumed-implied-spec, assumed-implied-spec-list*

C838 An implied-shape array shall be a **named constant**.

2 The **rank** of an implied-shape array is the number of *assumed-implied-specs* 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.

8.5.8.7 Assumed-rank entity

1 An assumed-rank entity is a **dummy data object** whose **rank** is assumed from its **effective argument**, or the associate name of a **RANK DEFAULT** block in a **SELECT RANK construct**; this rank can be zero. The bounds and shape of an assumed-rank entity with the **ALLOCATABLE** or **POINTER** attribute are determined as specified in 8.5.8.4. An assumed-rank entity is declared with an *array-spec* that is an *assumed-rank-spec*.

R828 *assumed-rank-spec* **is** ..

C839 An assumed-rank entity shall be an **associate name** or a **dummy data object** that does not have the **CODIMENSION** or **VALUE** attribute.

C840 An assumed-rank variable name shall not appear in a **designator** or expression except as an **actual argument** that corresponds to a **dummy argument** that is assumed-rank, the argument of the function **C_LOC** or **C_SIZEOF** from the intrinsic module **ISO_C_BINDING** (18.2), the first **dummy argument** of an **intrinsic inquiry function**, or the selector of a **SELECT RANK** statement.

C841 If an **assumed-size** or nonallocatable nonpointer assumed-rank array is an **actual argument** that corresponds to a **dummy argument** that is an **INTENT (OUT)** assumed-rank array, it shall not be **polymorphic**, **finalizable**, of a type with an **allocatable ultimate component**, or of a type for which **default initialization** is specified.

8.5.9 EXTERNAL attribute

1 The **EXTERNAL attribute** specifies that an entity is an **external procedure**, **dummy procedure**, **procedure pointer**, or **block data program unit**⁵.

C842 An entity shall not have both the **EXTERNAL attribute** and the **INTRINSIC attribute**.

C843 In an external subprogram, the **EXTERNAL attribute** shall not be specified for a procedure defined by the subprogram.

C844 In an interface body, the **EXTERNAL attribute** shall not be specified for the procedure declared by the interface body.

2 If an **external procedure** or **dummy procedure** is used as an **actual argument** or is the **target** of a procedure **pointer assignment**, it shall be declared to have the **EXTERNAL attribute**.

NOTE

The **EXTERNAL attribute** can be specified in a **type declaration statement**, by an **interface body** (15.4.3.2), by an **EXTERNAL statement** (15.4.3.5), or by a **procedure declaration statement** (15.4.3.6).

8.5.10 INTENT attribute

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**.

R829	<i>intent-spec</i>	is	IN
		or	OUT
		or	INOUT

C845 An entity with the **INTENT attribute** shall be a **dummy data object** or a dummy **procedure pointer**.

C846 (R829) A nonpointer object with the **INTENT (IN)** **attribute** shall not appear in a variable definition context (19.6.7).

C847 A pointer with the **INTENT (IN)** **attribute** shall not appear in a pointer association context (19.6.8).

C848 An **INTENT (OUT)** **dummy argument** of a nonintrinsic procedure shall not be an **allocatable coarray** or have a **subobject** that is an **allocatable coarray**.

⁵Block data program units are obsolescent.

- 1 C849 An entity with the **INTENT (OUT) attribute** shall not be of, or have a **subcomponent** of, type **EVENT_**
 2 **TYPE (16.10.2.10)**, **LOCK_TYPE (16.10.2.19)**, or **NOTIFY_TYPE (16.10.2.22)** from the intrinsic mod-
 3 **ule ISO_FORTRAN_ENV**.
- 4 2 The **INTENT (IN) attribute** for a nonpointer **dummy argument** specifies that it shall neither be defined nor
 5 become undefined during the invocation and execution of the procedure. The **INTENT (IN) attribute** for
 6 a pointer **dummy argument** specifies that during the invocation and execution of the procedure its asso-
 7 ciation shall not be changed except that it may become undefined if the **target** is deallocated other than
 8 through the pointer (**19.5.2.5**).
- 9 3 The **INTENT (OUT) attribute** for a nonpointer **dummy argument** specifies that the **dummy argument** be-
 10 comes undefined on invocation of the procedure, except for any subcomponents that are **default-initialized**
 11 (**7.5.4.6**). Any **actual argument** that corresponds to such a **dummy argument** shall be **definable**. The **INTENT**
 12 (**OUT**) **attribute** for a pointer **dummy argument** specifies that on invocation of the procedure the **pointer**
 13 **association** status of the **dummy argument** becomes undefined. Any **actual argument** that corresponds to
 14 such a dummy pointer shall be a pointer variable or a **procedure pointer** that is not the result of a function
 15 reference. Any undefinition or definition implied by association of an **actual argument** with an **INTENT**
 16 (**OUT**) **dummy argument** shall not affect any other entity within the statement that invokes the procedure.
- 17 4 The **INTENT (INOUT) attribute** for a nonpointer **dummy argument** specifies that any **actual argument** that
 18 corresponds to the **dummy argument** shall be **definable**. The **INTENT (INOUT) attribute** for a pointer
 19 **dummy argument** specifies that any **actual argument** that corresponds to the **dummy argument** shall be
 20 a pointer variable or a **procedure pointer** that is not the result of a function reference.

NOTE 1

The **INTENT attribute** for an **allocatable dummy argument** applies to both the allocation status and the definition status. An **actual argument** that corresponds to an **INTENT (OUT) allocatable dummy argument** is deallocated on procedure invocation (**9.7.3.2**). To avoid this deallocation for coarrays, **INTENT (OUT)** is not allowed for a **dummy argument** that is an allocatable coarray or has a subobject that is an allocatable coarray.

- 21 5 If no **INTENT attribute** is specified for a **dummy argument**, its use is subject to the limitations of its **effective**
 22 **argument (15.5.2)**.
- 23 6 If a nonpointer object has an **INTENT attribute**, then all of its subobjects have the same **INTENT attribute**.

NOTE 2

An example of **INTENT** specification is:

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

NOTE 3

If a **dummy argument** is a nonpointer derived-type object with a pointer component, then the pointer as a pointer is a subobject of the **dummy argument**, but the **target** of the pointer is not. Therefore, the restrictions on subobjects of the **dummy argument** apply to the pointer in contexts where it is used as a pointer, but not in contexts where it is dereferenced to indicate its **target**. For example, if X is a nonpointer **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**).

NOTE 3 (cont.)

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 4

Argument intent specifications serve several purposes in addition to documenting the intended use of [dummy arguments](#). A processor can check whether an [INTENT \(IN\) dummy argument](#) is used in a way that could redefine it. A slightly more sophisticated processor could check to see whether an [INTENT \(OUT\) dummy argument](#) could possibly be referenced before it is defined. If the procedure's [interface](#) is explicit, the processor can also verify that [actual arguments](#) corresponding to [INTENT \(OUT\)](#) or [INTENT \(INOUT\) dummy arguments](#) are [definable](#). A more sophisticated processor could use this information to optimize the translation of the referencing [scoping unit](#) by taking advantage of the fact that [actual arguments](#) corresponding to [INTENT \(IN\) dummy arguments](#) will not be changed and that any prior value of an [actual argument](#) corresponding to an [INTENT \(OUT\) dummy argument](#) will not be referenced and could thus be discarded.

[INTENT \(OUT\)](#) means that the value of the argument after invoking the procedure is entirely the result of executing that procedure. If an argument might not be redefined and it is desired to have the argument retain its value in that case, [INTENT \(OUT\)](#) cannot be used because it would cause the argument to become undefined; however, [INTENT \(INOUT\)](#) can be used, even if there is no explicit reference to the value of the dummy argument.

[INTENT \(INOUT\)](#) is not equivalent to omitting the [INTENT attribute](#). The [actual argument](#) corresponding to an [INTENT \(INOUT\) dummy argument](#) is always required to be [definable](#), while an [actual argument](#) corresponding to a [dummy argument](#) without an [INTENT attribute](#) need be [definable](#) only if the [dummy argument](#) is actually redefined.

8.5.11 INTRINSIC attribute

1 The [INTRINSIC attribute](#) specifies that the entity is an intrinsic procedure. The procedure name may be a generic name ([16.7](#)), a [specific name](#) ([16.8](#)), or both. Specific intrinsic procedure names are obsolescent.

2 If the [specific name](#) of an intrinsic procedure ([16.8](#)) is used as an [actual argument](#), the name shall be explicitly specified to have the [INTRINSIC attribute](#). Note that a specific intrinsic procedure listed in Table [16.3](#) is not permitted to be used as an [actual argument](#) ([C1534](#)).

C850 If the generic name of an intrinsic procedure is explicitly declared to have the [INTRINSIC attribute](#), and it is also the generic name of one or more [generic interfaces](#) ([15.4.3.2](#)) accessible in the same [scoping unit](#), the procedures in the interfaces and the generic intrinsic procedure shall all be functions or all be subroutines.

8.5.12 OPTIONAL attribute

1 The [OPTIONAL attribute](#) specifies that the [dummy argument](#) need not have an [effective argument](#) in a [reference](#) to the procedure ([15.5.2.13](#)).

C851 An entity with the [OPTIONAL attribute](#) shall be a [dummy argument](#).

NOTE

The intrinsic function [PRESENT](#) ([16.9.163](#)) can be used to determine whether an optional [dummy argument](#) has an associated [effective argument](#).

8.5.13 PARAMETER attribute

1 The [PARAMETER attribute](#) specifies that an entity is a [named constant](#). The entity has the value specified by its [constant-expr](#), converted, if necessary, to the type, type parameters and shape of the entity.

C852 An entity with the [PARAMETER attribute](#) shall not be a [variable](#), a [coarray](#), or a [procedure](#).

1 C853 An expression that specifies a **length type parameter** or array bound of a **named constant** shall be
2 a constant expression.

3 2 A **named constant** shall not be referenced unless it has been defined previously; it may be defined previ-
4 ously in the same statement.

NOTE

Examples of declarations with a **PARAMETER attribute** are:

```
REAL, PARAMETER :: ONE = 1.0, Y = 4.1 / 3.0
INTEGER, DIMENSION (3), PARAMETER :: ORDER = (/ 1, 2, 3 /)
TYPE (NODE), PARAMETER :: DEFAULT = NODE(0, NULL ( ))
```

8.5.14 POINTER attribute

6 1 Entities with the **POINTER attribute** can be associated with different data objects or procedures during
7 execution of a program. A pointer is either a **data pointer** or a **procedure pointer**.

8 C854 An entity with the **POINTER attribute** shall not have the **ALLOCATABLE**, **INTRINSIC**, or **TARGET**
9 attribute, and shall not be a **coarray**.

10 C855 A named procedure with the **POINTER attribute** shall have the **EXTERNAL attribute**.

11 2 A **data pointer** shall not be referenced unless it is **pointer associated** with a **target** object that is defined. A
12 **data pointer** shall not be defined unless it is **pointer associated** with a **target** object that is **definable**.

13 3 If a **data pointer** is associated, the values of its **deferred type parameters** are the same as the values of the
14 corresponding type parameters of its **target**.

15 4 A **procedure pointer** shall not be referenced unless it is **pointer associated** with a **target** procedure.

NOTE

Examples of **POINTER attribute** specifications are:

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

8.5.15 PROTECTED attribute

17 1 The **PROTECTED attribute** imposes limitations on the usage of module entities.

18 C856 The **PROTECTED attribute** shall be specified only in the specification part of a module.

19 C857 An entity with the **PROTECTED attribute** shall be a **procedure pointer** or variable.

20 C858 An entity with the **PROTECTED attribute** shall not be in a **common block**.⁶

21 C859 A nonpointer object that has the **PROTECTED attribute** and is accessed by **use association** shall not
22 appear in a variable definition context (19.6.7) or as a **data-target** or **initial-data-target**.

23 C860 A pointer that has the **PROTECTED attribute** and is accessed by **use association** shall not appear in
24 a pointer association context (19.6.8).

⁶Common blocks are obsolescent.

- 1 2 Other than within the module in which an entity is given the **PROTECTED attribute**, or within any of its
 2 **descendants**,
- 3 • if it is a nonpointer object, it is not **definable**, and
 - 4 • if it is a pointer, its association status shall not be changed except that it may become undefined if its
 5 **target** is deallocated other than through the pointer (19.5.2.5), or if its **target** becomes undefined by
 6 completing execution of a **BLOCK construct** or by execution of a **RETURN** or **END** statement.
- 7 3 If an object has the **PROTECTED attribute**, all of its subobjects have the **PROTECTED attribute**.

NOTE

An example of the **PROTECTED attribute**:

```

MODULE temperature
  REAL, PROTECTED :: temp_c, temp_f
CONTAINS
  SUBROUTINE set_temperature_c(c)
    REAL, INTENT(IN) :: c
    temp_c = c
    temp_f = temp_c*(9.0/5.0) + 32
  END SUBROUTINE
END MODULE

```

The **PROTECTED attribute** ensures that the variables `temp_c` and `temp_f` cannot be modified other than via the `set_temperature_c` procedure, thus keeping them consistent with each other.

8 8.5.16 SAVE attribute

- 9 1 The **SAVE attribute** specifies that a **local variable** of a **program unit** or subprogram retains its association
 10 status, allocation status, definition status, and value after execution of a **RETURN** or **END** statement unless
 11 it is a pointer and its **target** becomes undefined (19.5.2.5(6)). If it is a **local variable** of a subprogram it is
 12 shared by all instances (15.6.2.4) of the subprogram.
- 13 2 The **SAVE attribute** specifies that a **local variable** of a **BLOCK construct** retains its association status, alloca-
 14 tion status, definition status, and value after termination of the construct unless it is a pointer and its **target**
 15 becomes undefined (19.5.2.5(7)). If the **BLOCK construct** is within a subprogram the variable is shared by
 16 all instances (15.6.2.4) of the subprogram.
- 17 3 Giving a **common block** the **SAVE attribute** confers the attribute on all entities in the **common block**⁷.
- 18 C861 An entity with the **SAVE attribute** shall be a **common block**, variable, or **procedure pointer**.
- 19 C862 The **SAVE attribute** shall not be specified for a **dummy argument**, a function result, an **automatic**
 20 **data object**, or an object that is in a **common block**.
- 21 4 A variable, **common block**, or **procedure pointer** declared in the **scoping unit** of a main program, module, or
 22 submodule implicitly has the **SAVE attribute**, which may be confirmed by explicit specification. If a **common**
 23 **block** has the **SAVE attribute** in any other kind of **scoping unit**, it shall have the **SAVE attribute** in every
 24 **scoping unit** that is not of a main program, module, or submodule.

25 8.5.17 RANK clause

- 26 1 The RANK clause specifies the **DIMENSION attribute**.
- 27 R830 *rank-clause* **is** RANK (*scalar-int-constant-expr*)

⁷Common blocks are obsolescent.

- 1 C863 The *scalar-int-constant-expr* in a *rank-clause* shall be nonnegative with a value less than or equal to
2 the maximum array rank supported by the processor.
- 3 C864 An entity declared with a *rank-clause* shall be a *dummy data object* or have the *ALLOCATABLE* or
4 *POINTER* attribute.
- 5 2 An entity declared with a *RANK* clause has the specified rank. If the rank is zero the entity is scalar; other-
6 wise, if it has the *ALLOCATABLE* or *POINTER* attribute, it specifies that it is a *deferred-shape array*; other-
7 wise, it specifies that it is an *assumed-shape array* with all the lower bounds equal to one.

NOTE

Examples of RANK specifications are:

```

INTEGER :: X0(10,10,10)
LOGICAL, RANK(RANK(X0)), ALLOCATABLE :: X1 ! Rank 3, deferred shape
COMPLEX, RANK(2), POINTER :: X2           ! Rank 2, deferred-shape
LOGICAL, RANK(RANK(X0)) :: X3             ! Rank 3, assumed-shape
REAL, RANK(0) :: X4                       ! Scalar

```

8 8.5.18 TARGET attribute

- 9 1 The *TARGET attribute* specifies that a data object may have a *pointer associated* with it (10.2.2). An object
10 without the *TARGET attribute* shall not have a *pointer associated* with it.
- 11 C865 An entity with the *TARGET attribute* shall be a variable.
- 12 C866 An entity with the *TARGET attribute* shall not have the *POINTER attribute*.
- 13 2 If an object has the *TARGET attribute*, then all of its nonpointer subobjects also have the *TARGET attribute*.

NOTE 1

In addition to variables explicitly declared to have the *TARGET attribute*, the objects created by allocation of pointers (9.7.1.4) have the *TARGET attribute*.

NOTE 2

Examples of *TARGET attribute* specifications are:

```

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

```

NOTE 3

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*.

14 8.5.19 VALUE attribute

- 15 1 The *VALUE attribute* specifies a type of *argument association* (15.5.2.5) for a *dummy argument*.
- 16 C867 An entity with the *VALUE attribute* shall be a *dummy data object*. It shall not be an *assumed-size*
17 *array*, a *coarray*, or a variable with a *coarray potential subobject component*.
- 18 C868 An entity with the *VALUE attribute* shall not have the *ALLOCATABLE*, *INTENT (INOUT)*, *INTENT*
19 *(OUT)*, *POINTER*, or *VOLATILE* attributes.

1 C869 A **dummy argument** of a procedure with the **BIND attribute** shall not have both the **OPTIONAL** and
2 **VALUE** attributes.

3 **8.5.20 VOLATILE attribute**

4 1 The **VOLATILE attribute** specifies that an object may be referenced, defined, or become undefined, by means
5 not specified by the program. A pointer with the **VOLATILE attribute** may additionally have its association
6 status, **dynamic type** and type parameters, and array bounds changed by means not specified by the pro-
7 gram. An **allocatable** object with the **VOLATILE attribute** may additionally have its allocation status, **dy-**
8 **dynamic type** and type parameters, and array bounds changed by means not specified by the program.

9 C870 An entity with the **VOLATILE attribute** shall be a variable that is not an **INTENT (IN) dummy argu-**
10 **ment**.

11 C871 The **VOLATILE attribute** shall not be specified for a **coarray**, or a variable with a **coarray potential**
12 **subobject component**, that is accessed by **use** (14.2.2) or **host** (19.5.1.4) association.

13 C872 Within a **BLOCK construct** (11.1.4), the **VOLATILE attribute** shall not be specified for a **coarray**, or a
14 variable with a **coarray potential subobject component**, that is not a **construct entity** (19.4) of that
15 construct.

16 2 A noncoarray object that has the **VOLATILE attribute** may be associated with an object that does not have
17 the **VOLATILE attribute**, including by **use** (14.2.2) or **host** association (19.5.1.4). If an object that is not a
18 **local variable** of a **BLOCK construct** is specified to have the **VOLATILE attribute** in the *specification-part* of
19 the construct, the object has the attribute within the construct even if it does not have the attribute outside
20 the construct. The relationship between **coarrays**, the **VOLATILE attribute**, and **argument association** is
21 described in 15.5.2.9. The relationship between **coarrays**, the **VOLATILE attribute**, and **pointer association**
22 is described in 10.2.2.3.

23 3 A pointer should have the **VOLATILE attribute** if its target has the **VOLATILE attribute**. If, by means not
24 specified by the program, the target is referenced, defined, or becomes undefined, the pointer shall have
25 the **VOLATILE attribute**. All members of an EQUIVALENCE group should have the **VOLATILE attribute** if any
26 member has the **VOLATILE attribute**.⁸

27 4 If an object has the **VOLATILE attribute**, then all of its subobjects also have the **VOLATILE attribute**.

28 5 The Fortran processor should use the most recent definition of a volatile object each time its value is re-
29 quired. When a volatile object is defined by means of Fortran, it should make that definition available to
30 the non-Fortran parts of the program as soon as possible.

31 **8.6 Attribute specification statements**

32 **8.6.1 Accessibility statement**

33 R831 *access-stmt* is *access-spec* [[::] *access-id-list*]

34 R832 *access-id* is *access-name*
35 or *generic-spec*

36 C873 (R831) An *access-stmt* shall appear only in the *specification-part* of a module. Only one accessibility
37 statement with an omitted *access-id-list* is permitted in the *specification-part* of a module.

⁸EQUIVALENCE groups are obsolescent.

- 1 C874 (R832) Each *access-name* shall be the name of a module, variable, procedure, nonintrinsic type,
2 *named constant*, or namelist group.
- 3 C875 A module whose name appears in an *access-stmt* shall be referenced by a *USE statement* in the
4 *scoping unit* that contains the *access-stmt*.
- 5 C876 The name of a module shall appear at most once in all of the *access-stmts* in a module.
- 6 1 An *access-stmt* with an *access-id-list* specifies the *accessibility attribute*, *PUBLIC* or *PRIVATE*, of each *access-*
7 *id* in the list that is not a module name. An *access-stmt* without an *access-id* list specifies the default access-
8 ibility of the identifiers of entities declared in the module, and of entities accessed from a module whose
9 name does not appear in any *access-stmt* in the module. If an identifier is accessed from another module
10 and also declared locally, it has the default accessibility of a locally declared identifier. The statement
11 *PUBLIC*
12 specifies a default of public accessibility. The statement
13 *PRIVATE*
14 specifies a default of private accessibility. If no such statement appears in a module, the default is public
15 accessibility.
- 16 2 If an identifier is accessed by *use association* and not declared in the module, and the name of every module
17 from which it is accessed appears in an *access-stmt* in the scoping unit, its default accessibility is *PRIVATE*
18 if the *access-spec* in every such *access-stmt* is *PRIVATE*, or *PUBLIC* if the *access-spec* in any such *access-stmt*
19 is *PUBLIC*.

NOTE 1

Examples of accessibility statements are:

```
MODULE EX
  PRIVATE
  PUBLIC :: A, B, C, ASSIGNMENT (=), OPERATOR (+)
```

NOTE 2

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 from m1, and export our own procedures.
  PRIVATE m1
  PUBLIC m_type
  ... definitions for our own entities and module procedures.
END MODULE
```

20 8.6.2 ALLOCATABLE statement

- 21 R833 *allocatable-stmt* **is** *ALLOCATABLE* [::] *allocatable-decl-list*
- 22 R834 *allocatable-decl* **is** *object-name* [(*array-spec*)] ■
23 ■ [*lbracket coarray-spec rbracket*]
- 24 1 The *ALLOCATABLE* statement specifies the *ALLOCATABLE attribute* (8.5.3) for a list of objects.

NOTE

An example of an *ALLOCATABLE* statement is:

```
REAL A, B (:), SCALAR
ALLOCATABLE :: A (:, :), B, SCALAR
```

8.6.3 ASYNCHRONOUS statement

R835 *asynchronous-stmt* is ASYNCHRONOUS [::] *object-name-list*

- 1 The ASYNCHRONOUS statement specifies the [ASYNCHRONOUS attribute](#) (8.5.4) for a list of objects.

8.6.4 BIND statement

R836 *bind-stmt* is *language-binding-spec* [::] *bind-entity-list*

R837 *bind-entity* is *entity-name*
or */ common-block-name /*

C877 (R836) 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](#)⁹.

8.6.5 CODIMENSION statement

R838 *codimension-stmt* is CODIMENSION [::] *codimension-decl-list*

R839 *codimension-decl* is *coarray-name lbracket coarray-spec rbracket*

- 1 The CODIMENSION statement specifies the [CODIMENSION attribute](#) (8.5.6) for a list of objects.

NOTE

An example of a CODIMENSION statement is:

```
CODIMENSION a[*], b[3,*], c[:]
```

8.6.6 CONTIGUOUS statement

R840 *contiguous-stmt* is CONTIGUOUS [::] *object-name-list*

- 1 The CONTIGUOUS statement specifies the [CONTIGUOUS attribute](#) (8.5.7) for a list of objects.

8.6.7 DATA statement

R841 *data-stmt* is DATA *data-stmt-set* [[,] *data-stmt-set*] ...

- 1 The DATA statement specifies [explicit initialization](#) (8.4).
- 2 If a nonpointer variable has [default initialization](#), it shall not appear in a *data-stmt-object-list*.
- 3 A variable that appears in a DATA statement and has not been typed previously shall not appear in a subsequent type declaration unless 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](#)¹⁰, 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.

⁹Common blocks are obsolescent.

¹⁰Common blocks are obsolescent.

- 1 R842 *data-stmt-set* is *data-stmt-object-list* / *data-stmt-value-list* /
- 2 R843 *data-stmt-object* is *variable*
- 3 or *data-implied-do*
- 4 R844 *data-implied-do* is (*data-i-do-object-list* , [*integer-type-spec* ::] *data-i-do-variable* ■
- 5 ■ = *scalar-int-constant-expr* , *scalar-int-constant-expr* ■
- 6 ■ [, *scalar-int-constant-expr*])
- 7 R845 *data-i-do-object* is *array-element*
- 8 or *scalar-structure-component*
- 9 or *data-implied-do*
- 10 R846 *data-i-do-variable* is *do-variable*
- 11 C878 A *data-stmt-object* or *data-i-do-object* shall not be a *coindexed* variable.
- 12 C879 (R843) A *data-stmt-object* that is a *variable* shall be a *designator*. Each subscript, section subscript,
- 13 substring starting point, and substring ending point in the variable shall be a *constant expression*.
- 14 C880 (R843) A variable whose *designator* appears as a *data-stmt-object* or a *data-i-do-object* shall not
- 15 be a *dummy argument*, accessed by use or *host* association, in a named *common block* unless the
- 16 DATA statement is in a block data program unit, in *blank common*, a function name, a function result
- 17 name, an *automatic data object*, or an *allocatable* variable.
- 18 C881 (R843) A *data-i-do-object* or a *variable* that appears as a *data-stmt-object* shall not be an *object*
- 19 *designator* in which a pointer appears other than as the entire rightmost *part-ref*.
- 20 C882 (R845) The *array-element* shall be a variable.
- 21 C883 (R845) The *scalar-structure-component* shall be a variable.
- 22 C884 (R845) The *scalar-structure-component* shall contain at least one *part-ref* that contains a *subscript-*
- 23 *list*.
- 24 C885 (R845) In an *array-element* or *scalar-structure-component* that is a *data-i-do-object*, any subscript
- 25 shall be a *constant expression*, and any primary within that subscript that is a *data-i-do-variable*
- 26 shall be a DO variable of this *data-implied-do* or of a containing *data-implied-do*.
- 27 R847 *data-stmt-value* is [*data-stmt-repeat* *] *data-stmt-constant*
- 28 R848 *data-stmt-repeat* is *scalar-int-constant*
- 29 or *scalar-int-constant-subobject*
- 30 C886 (R848) The *data-stmt-repeat* shall be positive or zero. If the *data-stmt-repeat* is a *named constant*,
- 31 it shall have been defined previously.
- 32 R849 *data-stmt-constant* is *scalar-constant*
- 33 or *scalar-constant-subobject*
- 34 or *signed-int-literal-constant*
- 35 or *signed-real-literal-constant*
- 36 or *null-init*
- 37 or *initial-data-target*
- 38 or *structure-constructor*

or *enum-constructor*
or *enumeration-constructor*

C887 (R849) If a DATA statement constant value is a *named constant*, *structure constructor*, *enum constructor*, or *enumeration constructor*, the *named constant* or type shall have been defined previously.

C888 If a *data-stmt-constant* is a *structure-constructor*, *enum-constructor*, or *enumeration-constructor*, it shall be a *constant expression*.

R850 *int-constant-subobject* is *constant-subobject*

C889 (R850) *int-constant-subobject* shall be of type integer.

R851 *constant-subobject* is *designator*

C890 (R851) *constant-subobject* shall be a subobject of a constant.

C891 (R851) 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 (9.5.3.3). An *array section* is equivalent to the sequence of its array elements in array element order. A *data-implied-do* is expanded to form a sequence of array elements and *structure components*, under the control of the *data-i-do-variable*, as in the *DO construct* (11.1.7.4). The scope and attributes of a *data-i-do-variable* are described in 19.4.

6 The *data-stmt-value-list* is expanded to form a sequence of *data-stmt-constants*. A *data-stmt-repeat* indicates the number of times the following *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-constants* to the expanded sequence of scalar *data-stmt-constants*.

8 The expanded sequences of variables and *data-stmt-constants* 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 (7.5.4.6) with the initial data *target*; the data statement object is initially associated with the *target*.

- 1 10 A *data-stmt-constant* other than *boz-literal-constant*, *null-init*, or *initial-data-target* shall be compatible
 2 with its corresponding variable according to the rules of intrinsic assignment (10.2.1.2). The variable is
 3 initially defined with the value specified by the *data-stmt-constant*; if necessary, the value is converted ac-
 4 cording to the rules of intrinsic assignment (10.2.1.3) to a value that agrees in type, type parameters, and
 5 shape with the variable.
- 6 11 If a *data-stmt-constant* is a *boz-literal-constant*, the corresponding variable shall be of type integer. The *boz-*
 7 *literal-constant* is treated as if it were converted by the intrinsic function INT (16.9.110) to type integer with
 8 the kind type parameter of the variable.

NOTE

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 7.5.2.1, NOTE. The pointer HEAD_OF_LIST is declared using the derived type NODE from 7.5.4.6, NOTE 4; it is initially disassociated. MYNAME is initialized by a *structure constructor*. YOURNAME is initialized by supplying a separate value for each component.

9 8.6.8 DIMENSION statement

10 R852 *dimension-stmt* is DIMENSION [::] *array-name* (*array-spec*) ■
 11 ■ [, *array-name* (*array-spec*)] ...

- 12 1 The DIMENSION statement specifies the DIMENSION attribute (8.5.8) for a list of objects.

NOTE

An example of a DIMENSION statement is:

```
DIMENSION A (10), B (10, 70), C (:)
```

13 8.6.9 INTENT statement

14 R853 *intent-stmt* is INTENT (*intent-spec*) [::] *dummy-arg-name-list*

- 15 1 The INTENT statement specifies the INTENT attribute (8.5.10) for the *dummy arguments* in the list.

NOTE

An example of an INTENT statement is:

```
SUBROUTINE EX (A, B)
  INTENT (INOUT) :: A, B
```

16 8.6.10 OPTIONAL statement

17 R854 *optional-stmt* is OPTIONAL [::] *dummy-arg-name-list*

- 1 The OPTIONAL statement specifies the **OPTIONAL attribute** (8.5.12) for the **dummy arguments** in the list.

NOTE

An example of an OPTIONAL statement is:

```
SUBROUTINE EX (A, B)
  OPTIONAL :: B
```

8.6.11 PARAMETER statement

- 1 The PARAMETER statement specifies the **PARAMETER attribute** (8.5.13) and the values for the **named constants** in the list.

R855 *parameter-stmt* is PARAMETER (*named-constant-def-list*)

R856 *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 (8.7). A named array constant defined by a PARAMETER statement shall have its **rank** specified in a prior specification statement.

- 3 The constant expression that corresponds to a **named constant** shall have type and **type parameters** that conform with the **named constant** as specified for intrinsic assignment (10.2.1.2). If the **named constant** has implied shape, the expression shall have the same rank as the **named constant**; otherwise, the expression shall either be scalar or have the same shape as the **named constant**.

- 4 The value of each **named constant** is that specified by the corresponding **constant expression**; if necessary, the value is converted according to the rules of intrinsic assignment (10.2.1.3) to a value that agrees in type, type parameters, and shape with the **named constant**.

NOTE

An example of a PARAMETER statement is:

```
PARAMETER (MODULUS = MOD (28, 3), NUMBER_OF_SENATORS = 100)
```

8.6.12 POINTER statement

R857 *pointer-stmt* is POINTER [::] *pointer-decl-list*

R858 *pointer-decl* is *object-name* [(*deferred-shape-spec-list*)]

or *procptr-entity-name*

C892 A *procptr-entity-name* shall have the **EXTERNAL attribute**.

- 1 The POINTER statement specifies the **POINTER attribute** (8.5.14) for a list of entities.

NOTE

An example of a POINTER statement is:

```
TYPE (NODE) :: CURRENT
POINTER :: CURRENT, A (:, :)
```

8.6.13 PROTECTED statement

R859 *protected-stmt* is PROTECTED [::] *entity-name-list*

1 The PROTECTED statement specifies the **PROTECTED attribute** (8.5.15) for a list of entities.

8.6.14 SAVE statement

R860 *save-stmt* is SAVE [[::] *saved-entity-list*]

R861 *saved-entity* is *object-name*
or *proc-pointer-name*
or / *common-block-name* /

R862 *proc-pointer-name* is *name*

C893 (R860) 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**.

C894 A *proc-pointer-name* shall be the name of a **procedure pointer**.

1 A SAVE statement with a saved entity list specifies the **SAVE attribute** (8.5.16) for a list of entities¹¹. 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

An example of a SAVE statement is:

```
SAVE A, B, C, / BLOCKA /, D
```

8.6.15 TARGET statement

R863 *target-stmt* is TARGET [[::] *target-decl-list*]

R864 *target-decl* is *object-name* [(*array-spec*)] ■
■ [*lbracket coarray-spec rbracket*]

1 The TARGET statement specifies the **TARGET attribute** (8.5.18) for a list of objects.

NOTE

An example of a TARGET statement is:

```
TARGET :: A (1000, 1000), B
```

8.6.16 VALUE statement

R865 *value-stmt* is VALUE [[::] *dummy-arg-name-list*]

1 The VALUE statement specifies the **VALUE attribute** (8.5.19) for a list of **dummy arguments**.

8.6.17 VOLATILE statement

R866 *volatile-stmt* is VOLATILE [[::] *object-name-list*]

1 The VOLATILE statement specifies the **VOLATILE attribute** (8.5.20) for a list of objects.

¹¹Common blocks are obsolescent.

8.7 IMPLICIT statement

- 1 In a [scoping unit](#), an IMPLICIT statement specifies a type, and possibly type parameters, for all implicitly typed data entities whose names begin with one of the letters specified in the statement. An IMPLICIT NONE statement can indicate that no implicit typing rules are to apply in a particular [scoping unit](#), or that [external](#) and [dummy](#) procedures need to be explicitly given the [EXTERNAL attribute](#).

R867 *implicit-stmt* **is** IMPLICIT *implicit-spec-list*
or IMPLICIT NONE [([*implicit-none-spec-list*])]

R868 *implicit-spec* **is** *declaration-type-spec* (*letter-spec-list*)

R869 *letter-spec* **is** *letter* [– *letter*]

R870 *implicit-none-spec* **is** EXTERNAL
or TYPE

C895 (R867) If an IMPLICIT NONE statement appears in a [scoping unit](#), it shall precede any [PARAMETER statements](#) that appear in the [scoping unit](#). No more than one IMPLICIT NONE statement shall appear in a [scoping unit](#).

C896 The same *implicit-none-spec* shall not appear more than once in a given *implicit-stmt*.

C897 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](#).

C898 (R869) If the minus and second *letter* appear, the second letter shall follow the first letter alphabetically.

C899 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 have an [explicit interface](#) or be explicitly declared to have the [EXTERNAL attribute](#).

- 2 A *letter-spec* consisting of two *letters* separated by a minus is equivalent to writing a list containing all of the letters in alphabetical order in the alphabetic sequence from the first letter through the second letter. For example, A–C is equivalent to A, B, C. The same letter shall not appear as a single letter, or be included in a range of letters, more than once in all of the IMPLICIT statements in a [scoping unit](#).

- 3 In each [scoping unit](#), there is a mapping, which may be null, between each of the letters A, B, ..., Z and a type (and type parameters). An IMPLICIT statement specifies the mapping for the letters in its *letter-spec-list*. IMPLICIT NONE with an *implicit-none-spec* of TYPE or with no *implicit-none-spec-list* specifies the null mapping for all the letters. If a mapping is not specified for a letter, the default for a [program unit](#) or an [interface body](#) is default integer if the letter is I, J, ..., or N and default real otherwise, and the default for a [BLOCK construct](#), [internal subprogram](#), or [module subprogram](#) is the mapping in the [host scoping unit](#).

- 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. An explicit type specification in a [FUNCTION statement](#) overrides an IMPLICIT statement for the [result](#) of that function.

NOTE 1

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 2

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
  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 3

Implicit typing is not affected by **BLOCK constructs**. For example, in

```
SUBROUTINE S(N)
```

NOTE 3 (cont.)

```

...
IF (N>0) THEN
  BLOCK
    NSQP = CEILING (SQRT (DBLE (N)))
  END BLOCK
END IF
...
IF (N>0) THEN
  BLOCK
    PRINT *,NSQP
  END BLOCK
END IF
END SUBROUTINE

```

even if the only two appearances of `NSQP` are within the **BLOCK constructs**, the scope of `NSQP` is the whole subroutine `S`.

NOTE 4

In the subprogram

```

SUBROUTINE EXAMPLE (X, Y)
  IMPLICIT NONE (EXTERNAL)
  REAL, EXTERNAL :: G
  REAL :: X, Y
  X = F (Y)           ! Invalid: F lacks the EXTERNAL attribute.
  X = G (Y)           ! Valid: G has the EXTERNAL attribute.
END SUBROUTINE

```

the referenced function `F` needs to have the **EXTERNAL attribute** (8.5.9).

8.8 IMPORT statement

R871 *import-stmt* **is** `IMPORT [ [ :: ] import-name-list ]`
 or `IMPORT, ONLY : import-name-list`
 or `IMPORT, NONE`
 or `IMPORT, ALL`

C8100 (R871) An **IMPORT statement** shall not appear in the **scoping unit** of a *main-program*, *external-subprogram*, *module*, or *block-data*¹².

C8101 (R871) Each *import-name* shall be the name of an entity in the **host scoping unit**.

C8102 If any **IMPORT statement** in a **scoping unit** has an **ONLY** specifier, all **IMPORT statements** in that **scoping unit** shall have an **ONLY** specifier.

C8103 **IMPORT, NONE** shall not appear in the **scoping unit** of a submodule.

C8104 If an **IMPORT, NONE** or **IMPORT, ALL** statement appears in a **scoping unit**, no other **IMPORT statement** shall appear in that **scoping unit**.

C8105 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.

C8106 An entity whose name appears as an *import-name* or which is made accessible by an **IMPORT, ALL** statement shall not appear in any context described in 19.5.1.4 that would cause the host entity of that name to be inaccessible.

¹²Block data program units are obsolescent.

- 1 1 If the **ONLY** specifier appears on an IMPORT statement in a **scoping unit** other than a **BLOCK construct**, an
 2 entity is only accessible by host association if its name appears as an *import-name* in that **scoping unit**. If
 3 a **BLOCK construct** contains one or more IMPORT statements with ONLY specifiers, identifiers of local and
 4 **construct** entities in the **host scoping unit** that are not in the *import-name-list* of at least one of the IMPORT
 5 statements are inaccessible in the **BLOCK construct**.
- 6 2 An IMPORT, NONE statement in a **scoping unit** specifies that no entities in the **host scoping unit** are access-
 7 ible by host association in that **scoping unit**. This is the default for an interface body that is not a module
 8 procedure interface body. An IMPORT, NONE statement in a **BLOCK construct** specifies that the identifiers
 9 of local and **construct** entities in the **host scoping unit** are inaccessible in the **BLOCK construct**.
- 10 3 An IMPORT, ALL statement in a **scoping unit** specifies that all entities from the **host scoping unit** are access-
 11 ible in that **scoping unit**.
- 12 4 If an IMPORT statement with no specifier and no *import-name-list* appears in a **scoping unit**, every entity in
 13 the **host scoping unit** is accessible unless its name appears in a context described in 19.5.1.4 that causes it
 14 to be inaccessible. This is the default for a **derived-type definition**, **internal subprogram**, module procedure
 15 interface body, **module subprogram**, or **submodule**.
- 16 5 If an IMPORT statement with an *import-name-list* appears in a **scoping unit** other than a **BLOCK construct**,
 17 each entity named in the list is accessible.

NOTE 1

The IMPORT, NONE statement 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 2

The IMPORT, ALL statement can be used to prevent accidental “shadowing” of host entities:

```
SUBROUTINE outer
  REAL x
  ...
  CONTAINS
    SUBROUTINE inner
      IMPORT, ALL
      ...
      x = x + 1 ! There is a host X, so this must be the host X.
```

NOTE 3

The IMPORT, **ONLY** statement can be used to document deliberate access via host association whilst blocking accidental access:

```
SUBROUTINE sub
  IMPORT, ONLY : x, y
  ...
  x = y + z ! Only X and Y are imported, so Z is local.
```

NOTE 4

The program

```
PROGRAM MAIN
  BLOCK
    IMPORT, NONE
    !IMPORT, ONLY: X
    X = 1.0
```

NOTE 4 (cont.)

```
END BLOCK
END
```

is not conformant. The variable X is implicitly declared in the [scoping unit](#) of the main program. The statement IMPORT, NONE makes X inaccessible in the [BLOCK construct](#). If the IMPORT, NONE statement is replaced with the IMPORT statement in the comment, the program is conformant.

NOTE 5

The IMPORT statement can be used to allow [module procedures](#) to have [dummy arguments](#) that are procedures with [assumed-shape](#) arguments of an opaque type. For example:

```
MODULE M
  TYPE T
    PRIVATE      ! T is an opaque type
    ...
  END TYPE
CONTAINS
  SUBROUTINE PROCESS(X,Y,RESULT,MONITOR)
    TYPE(T), INTENT(IN) :: X(:, :), Y(:, :)
    TYPE(T), INTENT(OUT) :: RESULT(:, :)
    INTERFACE
      SUBROUTINE MONITOR(ITERATION_NUMBER, CURRENT_ESTIMATE)
        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.

8.9 NAMELIST statement

1

2 1 A NAMELIST statement specifies a group of named data objects, which can be referred to by a single name
3 for the purpose of data transfer ([12.6](#), [13.11](#)).

4 R872 *namelist-stmt* is NAMELIST ■
5 ■ / *namelist-group-name* / *namelist-group-object-list* ■
6 ■ [[,] / *namelist-group-name* / ■
7 ■ *namelist-group-object-list*] ...

8 C8107 (R872) The *namelist-group-name* shall not be a name accessed by [use association](#).

9 R873 *namelist-group-object* is [variable-name](#)

10 C8108 (R873) A *namelist-group-object* shall not be an [assumed-size array](#).

11 C8109 A *namelist-group-object* shall not be of enumeration type, or have a direct component that is of
12 enumeration type.

13 2 The order in which the values appear on output is the same as the order of the [namelist-group-objects](#) in
14 the namelist group object list; if a variable appears more than once as a [namelist-group-object](#) for the same
15 namelist group, its value appears once for each occurrence.

16 3 Any *namelist-group-name* may occur more than once in the NAMELIST statements in a [scoping unit](#). The
17 [namelist-group-object-list](#) following each successive appearance of the same *namelist-group-name* in a [scop-](#)

- 1 [ing unit](#) is treated as a continuation of the list for that *namelist-group-name*.
- 2 4 A namelist group object may be a member of more than one namelist group.
- 3 5 A namelist group object shall either be accessed by use or host association or shall have its [declared type](#),
 4 [kind type parameters](#) of the [declared type](#), and [rank](#) specified by previous statements in the same [scoping](#)
 5 [unit](#) or by the [implicit typing rules](#) in effect for the [scoping unit](#). If a namelist group object is typed by
 6 the [implicit typing rules](#), its appearance in any subsequent [type declaration statement](#) shall confirm the
 7 implied type and [type parameters](#).

NOTE

An example of a NAMELIST statement is:

```
NAMELIST /NLIST/ A, B, C
```

8 8.10 Storage association of data objects (obsolescent)

9 8.10.1 Obsolescence

- 10 1 Storage association of data objects, whether by the [EQUIVALENCE statement](#), [common blocks](#), or the [ENTRY](#)
 11 [statement](#), is obsolescent.

12 8.10.2 EQUIVALENCE statement

13 8.10.2.1 General

- 14 1 An EQUIVALENCE statement is used to specify the sharing of [storage units](#) by two or more objects in a
 15 [scoping unit](#). This causes [storage association](#) (19.5.3) of the objects that share the [storage units](#).
- 16 2 If the equivalenced objects have differing type or type parameters, the EQUIVALENCE statement does not
 17 cause type conversion or imply mathematical equivalence. If a scalar and an array are equivalenced, the
 18 scalar does not have array properties and the array does not have the properties of a scalar.

19 R874 *equivalence-stmt* **is** EQUIVALENCE *equivalence-set-list*

20 R875 *equivalence-set* **is** (*equivalence-object* , *equivalence-object-list*)

21 R876 *equivalence-object* **is** *variable-name*
 22 **or** *array-element*
 23 **or** *substring*

24 C8110 (R876) An *equivalence-object* shall not be a [designator](#) with a [base object](#) that is a [dummy argu-](#)
 25 [ment](#), a [function result](#), a pointer, an [allocatable](#) variable, a derived-type object that has an [allocat-](#)
 26 [able](#) or pointer [ultimate component](#), an object of a nonsequence derived type, an object of enumer-
 27 ation type, an [automatic data object](#), a [coarray](#), a variable with the [BIND attribute](#), a variable in a
 28 [common block](#) that has the [BIND attribute](#), or a [named constant](#).

29 C8111 (R876) An *equivalence-object* shall not be a [designator](#) that has more than one *part-ref*.

30 C8112 (R876) An *equivalence-object* shall not have the [TARGET attribute](#).

31 C8113 (R876) Each subscript or substring range expression in an *equivalence-object* shall be an integer
 32 [constant expression](#) (10.1.12).

- 1 C8114 (R875) If an *equivalence-object* is default integer, default real, double precision real, default com-
 2 plex, default logical, or of *numeric sequence type*, all of the objects in the equivalence set shall be
 3 of these types and kinds.
- 4 C8115 (R875) If an *equivalence-object* is default character or of *character sequence type*, all of the objects
 5 in the equivalence set shall be of these types and kinds.
- 6 C8116 (R875) If an *equivalence-object* is of a *sequence type* that is not a *numeric sequence* or *character*
 7 *sequence* type, all of the objects in the equivalence set shall be of that type.
- 8 C8117 (R875) If an *equivalence-object* is of an intrinsic type but is not default integer, default real, double
 9 precision real, default complex, default logical, or default character, all of the objects in the equival-
 10 ence set shall be of the same type with the same kind type parameter value.
- 11 C8118 (R876) If an *equivalence-object* has the *PROTECTED attribute*, all of the objects in the equivalence
 12 set shall have the *PROTECTED attribute*.
- 13 C8119 (R876) The name of an *equivalence-object* shall not be a name made accessible by *use association*.
- 14 C8120 (R876) A *substring* shall not have length zero.

NOTE

The EQUIVALENCE statement allows the equivalencing of sequence structures and the equivalencing of objects of intrinsic type with nondefault type parameters, but there are strict rules regarding the appearance of these objects in an EQUIVALENCE statement.

In addition to the above constraints, further rules on the interaction of EQUIVALENCE statements and *default initialization* are given in 19.5.3.4.

8.10.2.2 Equivalence association

- 1 An EQUIVALENCE statement specifies that the *storage sequences* (19.5.3.2) of the data objects specified
 2 in an *equivalence-set* are storage associated. All of the nonzero-sized sequences in the *equivalence-set*, if
 3 any, have the same first *storage unit*, and all of the zero-sized sequences in the *equivalence-set*, if any, are
 4 storage associated with one another and with the first *storage unit* of any nonzero-sized sequences. This
 5 causes the *storage association* of the data objects in the *equivalence-set* and can cause *storage association*
 6 of other data objects.
- 7 If any data object in an *equivalence-set* has the *SAVE attribute*, all other objects in the *equivalence-set* have
 8 the *SAVE attribute*; this may be confirmed by explicit specification.

8.10.2.3 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
 2 of a *character sequence type*. The lengths of equivalenced default character objects need not be the same.
- 3 An EQUIVALENCE statement specifies that the *storage sequences* of all the default character data objects
 4 specified in an *equivalence-set* are storage associated. All of the nonzero-sized sequences in the *equivalence-*
 5 *set*, if any, have the same first *character storage unit*, and all of the zero-sized sequences in the *equivalence-*
 6 *set*, if any, are storage associated with one another and with the first *character storage unit* of any nonzero-
 7 sized sequences. This causes the *storage association* of the data objects in the *equivalence-set* and can cause
 8 *storage association* of other data objects.

NOTE

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 ---	---	---			
---	C (1) ---	---	--- B ---	---	---	---
			---	C (2) ---	---	

8.10.2.4 Array names and array element designators

- 1 For a nonzero-sized array, the use of the array name unqualified by a subscript list as an *equivalence-object* has the same effect as using an array element *designator* that identifies the first element of the array.

8.10.2.5 Restrictions on EQUIVALENCE statements

- 1 An EQUIVALENCE statement shall not specify that the same *storage unit* is to occur more than once in a *storage sequence*.
- 2 An EQUIVALENCE statement shall not specify that consecutive *storage units* are to be nonconsecutive.

8.10.3 COMMON statement**8.10.3.1 General**

- 1 The COMMON statement specifies blocks of physical storage, called *common blocks*, that can be accessed by any of the *scoping units* in a program. Thus, the COMMON statement provides a global data facility based on *storage association* (19.5.3).
- 2 A *common block* that does not have a name is called *blank common*.

R877 *common-stmt* is COMMON ■
 ■ [/ [*common-block-name*] /] *common-block-object-list* ■
 ■ [[,] / [*common-block-name*] / ■
 ■ *common-block-object-list*] ...

R878 *common-block-object* is *variable-name* [(*array-spec*)]

C8121 (R878) An *array-spec* in a *common-block-object* shall be an *explicit-shape-spec-list*.

C8122 (R878) Only one appearance of a given *variable-name* is permitted in all *common-block-object-lists* within a *scoping unit*.

C8123 (R878) 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*, an *object of enumeration type*, a *procedure pointer*, an *automatic data object*, a *variable with the BIND attribute*, an *unlimited polymorphic pointer*, or a *coarray*.

C8124 (R878) 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*.

C8125 (R878) 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 (7.5.2.3)** appear in **common**, it is as if the individual components were enumerated directly in the common list.

8.10.3.2 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 (19.5.3.2)** of all data objects in the common block object lists for the **common block**. The order of the **storage sequences** is the same as the order of the appearance of the common block object lists in the **scoping unit**.
 - (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**.

8.10.3.3 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.

8.10.3.4 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 can 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.

- 1 3 A nonpointer object that is default character or of [character sequence type](#) shall be associated only with
2 nonpointer objects of these types and kinds.
- 3 4 A nonpointer object of a derived type that is not a [numeric sequence](#) or [character sequence](#) type shall be
4 associated only with nonpointer objects of the same type.
- 5 5 A nonpointer object of an enum type shall be associated only with nonpointer objects of the same type.
- 6 6 A nonpointer object of intrinsic type but which is not default integer, default real, double precision real,
7 default complex, default logical, or default character shall be associated only with nonpointer objects of the
8 same type and type parameters.
- 9 7 A [data pointer](#) shall be storage associated only with [data pointers](#) of the same type and [rank](#). [Data pointers](#)
10 that are storage associated shall have [deferred](#) the same type parameters; corresponding nondeferred [type](#)
11 [parameters](#) shall have the same value.
- 12 8 An object with the [TARGET attribute](#) shall be storage associated only with another object that has the [TAR-](#)
13 [GET attribute](#) and the same type and type parameters.

NOTE

A [common block](#) is permitted to contain sequences of different [storage units](#), provided each [scoping unit](#) that accesses the [common block](#) specifies an identical sequence of [storage units](#) for the [common block](#). For example, this allows a single [common block](#) to contain both [numeric](#) and [character storage units](#).

Association in different [scoping units](#) between objects of default type, objects of double precision real type, and sequence structures is permitted according to the rules for equivalence objects ([8.10.2](#)).

14 8.10.3.5 Differences between named common and blank common

- 15 1 A [blank common](#) block has the same properties as a named [common block](#), except for the following.
16
 - Execution of a [RETURN](#) or [END](#) statement might cause data objects in a named [common block](#) to
17 become undefined unless the [common block](#) has the [SAVE attribute](#), but never causes nonpointer
18 data objects in [blank common](#) to become undefined ([19.6.6](#)).
 - Named [common blocks](#) of the same name shall be of the same size in all [scoping units](#) of a program
19 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
20 [type declaration statement](#) in a [block data program unit](#) ([14.3](#)), but objects in [blank common](#) shall
21 not be initially defined.

24 8.10.4 Restrictions on common and equivalence

- 25 1 An [EQUIVALENCE statement](#) shall not cause the [storage sequences](#) of two different [common blocks](#) to be
26 associated.
- 27 2 Equivalence association shall not cause a derived-type object with [default initialization](#) to be associated
28 with an object in a [common block](#).
- 29 3 Equivalence association shall not cause a common block [storage sequence](#) to be extended by adding [storage](#)
30 [units](#) preceding the first [storage unit](#) of the first object specified in a COMMON statement for the [common](#)
31 [block](#).

9 Use of data objects

9.1 Designator

R901 *designator* is *object-name*
 or *array-element*
 or *array-section*
 or *coindexed-named-object*
 or *complex-part-designator*
 or *structure-component*
 or *substring*

1 The appearance of a *data object designator* in a context that requires its value is termed a reference.

9.2 Variable

R902 *variable* is *designator*
 or *function-reference*

C901 (R902) *designator* shall not be a constant or a subobject of a constant.

C902 (R902) *function-reference* shall have a *data pointer* result.

1 A variable is either the data object denoted by *designator* or the *target* of the pointer resulting from the evaluation of *function-reference*; this pointer shall be associated.

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 variable becomes defined with a value when events described in 19.6.5 occur.

R903 *variable-name* is *name*

C903 (R903) *variable-name* shall be the name of a variable.

R904 *logical-variable* is *variable*

C904 (R904) *logical-variable* shall be of type logical.

R905 *char-variable* is *variable*

C905 (R905) *char-variable* shall be of type character.

R906 *default-char-variable* is *variable*

C906 (R906) *default-char-variable* shall be default character.

R907 *int-variable* is *variable*

C907 (R907) *int-variable* shall be of type integer.

NOTE

For example, given the declarations:

```
CHARACTER (10) A, B (10)
TYPE (PERSON) P      ! See 7.5.2.1, NOTE
```

then A, B, B (1), B (1:5), P % AGE, and A (1:1) are all variables.

9.3 Constants

- 1 A constant (6.2.3) is a literal constant or a **named constant**. A literal constant is a scalar denoted by a syntactic form, which indicates its type, type parameters, and value. A **named constant** is a constant that has a name; the name has the **PARAMETER attribute** (8.5.13, 8.6.11). A reference to a constant is always permitted; redefinition of a constant is never permitted.

9.4 Scalars**9.4.1 Substrings**

- 1 A substring is a **contiguous** portion of a character string (7.4.4).

R908 *substring* is *parent-string* (*substring-range*)

R909 *parent-string* is *scalar-variable-name*
or *array-element*
or *coindexed-named-object*
or *scalar-structure-component*
or *scalar-constant*

R910 *substring-range* is [*scalar-int-expr*] : [*scalar-int-expr*]

C908 (R909) *parent-string* shall be of type character.

- 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 $\text{MAX}(l - f + 1, 0)$, 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

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

9.4.2 Structure components

- 1 A **structure component** is part of an object of derived type; it can be referenced by an **object designator**. A **structure component** may be a scalar or an array.

R911 *data-ref* is *part-ref* [% *part-ref*] ...

R912 *part-ref* is *part-name* [(*section-subscript-list*)] [*image-selector*]

C909 (R911) Each *part-name* except the rightmost shall be of derived type.

C910 (R911) Each *part-name* except the leftmost shall be the name of a component of the **declared type** of the preceding *part-name*.

C911 (R911) If the rightmost *part-name* is of **abstract type**, *data-ref* shall be **polymorphic**.

C912 (R911) The leftmost *part-name* shall be the name of a data object.

C913 (R912) If a *section-subscript-list* appears, the sum of the **rank** of *part-ref*, the sizes of the arrays in each multiple subscript, and the number of **subscripts**, shall equal the **rank** of *part-name*.

C914 (R912) If *image-selector* appears, the number of **cosubscripts** shall be equal to the **corank** of *part-name*.

C915 A *data-ref* shall not be of type **C_PTR** or **C_FUNPTR** from the intrinsic module **ISO_C_BINDING** (18.2), or of type **TEAM_TYPE** from the intrinsic module **ISO_FORTRAN_ENV** (16.10.2), if one of its *part-ref*s has an *image-selector*.

C916 (R912) If *image-selector* appears and *part-name* is an array, *section-subscript-list* shall appear.

C917 (R911) Except as an **actual argument** to an intrinsic **inquiry function** or as the *designator* in a type parameter inquiry, a *data-ref* shall not be a **coindexed object** that has a **polymorphic allocatable potential subobject component**.

C918 Except as an **actual argument** to an intrinsic **inquiry function** or as the *designator* in a type parameter inquiry, if the rightmost *part-ref* is **polymorphic**, no other *part-ref* shall be **coindexed**.

- 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 sum of the number of subscript triplets, the number of **vector subscripts**, and the sizes of one of the arrays in each multiple section subscript.

C919 (R911) There shall not be more than one *part-ref* with nonzero **rank**. A *part-name* to the right of a *part-ref* with nonzero **rank** shall not have the **ALLOCATABLE** or **POINTER** attribute.

- 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-names*, except for possibly the rightmost, is a pointer. If the rightmost *part-name* is the only pointer, then the *data-ref* is a subobject of its **base object** in contexts that pertain to its **pointer association** status but not in any other contexts.

NOTE 1

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.

1 R913 *structure-component* is *data-ref*

2 C920 (R913) There shall be more than one *part-ref* and the rightmost *part-ref* shall not have a *section-*
3 *subscript-list*.

4 6 A *structure component* shall be neither referenced nor defined before the declaration of the *base object*. A
5 *structure component* is a pointer only if the rightmost part name has the **POINTER** attribute.

NOTE 2

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

For a more elaborate example see [C.5.1](#).

NOTE 3

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 9.4.3 Coindexed named objects

7 1 A *coindexed-named-object* is a named scalar *coarray* variable followed by an image selector.

8 R914 *coindexed-named-object* is *data-ref*

9 C921 (R914) The *data-ref* shall contain exactly one *part-ref*. The *part-ref* shall contain an *image-selector*.
10 The *part-name* shall be the name of a scalar coarray.

11 9.4.4 Complex parts

12 R915 *complex-part-designator* is *designator* % RE
13 or *designator* % IM

14 C922 (R915) The *designator* shall be of complex type.

15 1 If *complex-part-designator* is *designator*%RE it designates the real part of *designator*. If it is *designator*%IM
16 it designates the imaginary part of *designator*. The type of a *complex-part-designator* is real, and its kind
17 and shape are those of the *designator*, which can be an *array* or *scalar*.

NOTE

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.

9.4.5 Type parameter inquiry

A **type parameter inquiry** is used to inquire about a type parameter of a data object. It applies to both intrinsic and derived types.

R916 *type-param-inquiry* is *designator* % *type-param-name*

C923 (R916) The *type-param-name* shall be the name of a type parameter of the **declared type** of the object designated by the *designator*.

A **deferred type parameter** of a pointer that is not associated or of an unallocated **allocatable** variable shall not be inquired about.

NOTE 1

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.

The intrinsic type parameters can also be inquired about by using the intrinsic functions **KIND** and **LEN**.

NOTE 2

The following are examples of type parameter inquiries:

a%kind
s%len
b(10)%kind
p%dim

A is real. Same value as **KIND** (a).
S is character. Same value as **LEN** (s).
Inquiry about an array element.
P is of the derived type `general_point`.

See 7.5.3.1, NOTE for the definition of the `general_point` type used in the last example above.

9.5 Arrays

9.5.1 Order of reference

No order of reference to the elements of an array is indicated by the appearance of the array **designator**, except where array element ordering (9.5.3.3) is specified.

9.5.2 Whole arrays

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.

The appearance of a **whole array** variable in an executable construct specifies all the elements of the array (5.4.6). The appearance of a whole array **designator** in a nonexecutable statement specifies the entire array except for the appearance of a **whole array** designator in an equivalence set¹ (8.10.2.4). An **assumed-size array** (8.5.8.5) is permitted to appear as a **whole array** in an executable construct or **specification expression** only as an **actual argument** in a **procedure reference** that does not require the shape.

9.5.3 Array elements and array sections

9.5.3.1 Syntax

R917 *array-element* is *data-ref*

¹Equivalence sets are obsolescent.

- 1 C924 (R917) Every *part-ref* shall have *rank* zero and the last *part-ref* shall contain a *subscript-list*.
- 2 R918 *array-section* is *data-ref* [(*substring-range*)]
- 3 or *complex-part-designator*
- 4 C925 (R918) Exactly one *part-ref* shall have nonzero *rank*, and either the final *part-ref* shall have a
- 5 *section-subscript-list* with nonzero *rank*, another *part-ref* shall have nonzero *rank*, or the *complex-*
- 6 *part-designator* shall be an array.
- 7 C926 (R918) If a *substring-range* appears, *data-ref* shall be of type character.
- 8 R919 *subscript* is *scalar-int-expr*
- 9 R920 *multiple-subscript* is @ *int-expr*
- 10 C927 The *int-expr* in a *multiple-subscript* shall be an array of rank one.
- 11 R921 *section-subscript* is *subscript*
- 12 or *multiple-subscript*
- 13 or *subscript-triplet*
- 14 or *multiple-subscript-triplet*
- 15 or *vector-subscript*
- 16 R922 *subscript-triplet* is [*subscript*] : [*subscript*] [: *stride*]
- 17 R923 *multiple-subscript-triplet* is @ [*int-expr*] : [*int-expr*] [: *int-expr*]
- 18 C928 A *multiple-subscript-triplet* shall have at least one *int-expr* that is an array of rank one. The *int-exprs*
- 19 in a *multiple-subscript-triplet* shall be conformable.
- 20 R924 *stride* is *scalar-int-expr*
- 21 R925 *vector-subscript* is *int-expr*
- 22 C929 (R925) A *vector-subscript* shall be an integer array expression of *rank* one.
- 23 C930 (R922) The second subscript shall not be omitted from a *subscript-triplet* in the last dimension of
- 24 an *assumed-size array*.
- 25 C931 If a *multiple-subscript-triplet* is the last *section-subscript* in the *section-subscript-list* of an assumed-
- 26 size array, the second *int-expr* shall appear.
- 27 1 An array element is a scalar. An *array section* is an array. If a *substring-range* appears in an *array-section*,
- 28 each element is the designated substring of the corresponding element of the *array section*.
- 29 2 The value of a subscript in an array element shall be within the bounds for its dimension.

NOTE 1

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 2

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 3

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

9.5.3.2 Sequences of subscripts and subscript triplets

- 1 A [multiple-subscript](#) specifies a sequence of subscripts, the number of which is equal to the size of [multiple-subscript](#). The effect is as if the array elements were specified individually as subscripts of consecutive dimensions (not preceded by @).
- 2 In a [multiple-subscript-triplet](#), if the first [int-expr](#) does not appear, the effect is as if it were a rank-one array whose element values are the lower bounds of the corresponding dimensions. If the second [int-expr](#) does not appear, the effect is as if it were a rank-one array whose element values are the upper bounds of the corresponding dimensions. If the third [int-expr](#) does not appear, the effect is as if it appeared with the value one.
- 3 A [multiple-subscript-triplet](#) specifies a sequence of subscript triplets, the number of which is equal to the size of one of its array [int-exprs](#). If any [int-expr](#) is a scalar, the effect is as if it were broadcast to the shape of one that is an array. An element of the first array acts as if it were the first [subscript](#) in a [subscript](#) triplet; the corresponding element of the second array acts as if it were the second [subscript](#); the corresponding element of the third array acts as if it were the [stride](#).

NOTE

Examples of references to parts of arrays using one-dimensional arrays to specify sequences of subscripts or sequences of subscript triplets, assuming V1, V2, and V3 are rank-one arrays, are:

A(@[3,5])	! Array element, equivalent to A(3, 5)
A(6, @[3,5], 1)	! Array element, equivalent to A(6, 3, 5, 1)
A(@[1,2]:[3,4])	! Array section, equivalent to A(1:3, 2:4)
A(@:[4,6]:2, :, 1)	! Array section with stride, equivalent to A(:4:2, :6:2, :, 1)
A(@V1, :, @V2)	! Rank-one array section, the rank of A being ! SIZE (V1) + 1 + SIZE (V2).
B(@V1, :, @V2:)	! Rank 1 + SIZE (V2) array section, the rank of B being ! SIZE (V1) + 1 + SIZE (V2).
C(@V1, :, @::V3)	! Rank 1 + SIZE (V3) array section, the rank of C being ! SIZE (V1) + 1 + SIZE (V3).

9.5.3.3 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 9.1.

Table 9.1 — 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	$1 + (s_1 - j_1) + (s_2 - j_2) \times d_1$

Subscript order value (cont.)

Rank	Subscript bounds	Subscript list	Subscript order value
3	$j_1:k_1, j_2:k_2, j_3:k_3$	s_1, s_2, s_3	$1 + (s_1 - j_1)$ $+ (s_2 - j_2) \times d_1$ $+ (s_3 - j_3) \times d_2 \times d_1$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
15	$j_1:k_1, \dots, j_{15}:k_{15}$	$s_1, \dots, s_{15}$	$1 + (s_1 - j_1)$ $+ (s_2 - j_2) \times d_1$ $+ \dots$ $+ (s_{15} - j_{15}) \times d_{14} \times \dots \times d_1$
NOTE 1 $d_i = \max(k_i - j_i + 1, 0)$ is the size of the i^{th} dimension.			
NOTE 2 If the size of the array is nonzero, $j_i \leq s_i \leq k_i$ for all $i = 1, 2, \dots, 15$.			

9.5.3.4 Array sections

9.5.3.4.1 Section subscript lists

- 1 In an *array-section* having a *section-subscript-list*, each subscript triplet and *vector-subscript* in the section subscript list indicates a sequence of subscripts, which may be empty. Each subscript in such a sequence shall be within the bounds for its dimension unless the sequence is empty. The *array section* is the set of elements from the array determined by all possible subscript lists obtainable from the single subscripts or sequences of subscripts specified by each section subscript.
- 2 In an *array-section* with no *section-subscript-list*, the *rank* and shape of the array is the *rank* and shape of the *part-ref* with nonzero *rank*; otherwise, the *rank* of the *array section* is the number of subscript triplets and *vector subscripts* in the section subscript list. The shape is the rank-one array whose i^{th} element is the number of integer values in the sequence indicated by the i^{th} subscript triplet or *vector subscript*. If any of these sequences is empty, the *array section* has size zero. The subscript order of the elements of an *array section* is that of the array data object that the *array section* represents.

9.5.3.4.2 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 1

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

- 1 the stride down to the smallest such integer equal to or greater than the second subscript; the sequence is
2 empty if the second subscript is greater than the first.

NOTE 2

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 3

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.

3 9.5.3.4.3 Vector subscript

- 4 1 A [vector subscript](#) designates a sequence of subscripts corresponding to the values of the elements of the
5 expression. Each element of the expression shall be defined.
- 6 2 An [array section](#) with a [vector subscript](#) shall not be [finalized](#) by a nonelemental [final subroutine](#).
- 7 3 If a [vector subscript](#) has two or more elements with the same value, an [array section](#) with that [vector sub-](#)
8 [script](#) is not [definable](#) and shall not be [defined](#) or become [undefined](#).

NOTE

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:

$$\begin{aligned} U &= [1, 3, 2] \\ V &= [2, 1, 1, 3] \end{aligned}$$

Then Z (3, V) consists of elements from the third row of Z in the order:

$$Z (3, 2) \quad Z (3, 1) \quad Z (3, 1) \quad Z (3, 3)$$

Z (U, 2) consists of the column elements:

$$Z (1, 2) \quad Z (3, 2) \quad Z (2, 2)$$

and Z (U, V) consists of the elements:

$$\begin{array}{cccc} Z (1, 2) & Z (1, 1) & Z (1, 1) & Z (1, 3) \\ Z (3, 2) & Z (3, 1) & Z (3, 1) & Z (3, 3) \\ Z (2, 2) & Z (2, 1) & Z (2, 1) & Z (2, 3) \end{array}$$

Because Z (3, V) and Z (U, V) contain duplicate elements from Z, the sections Z (3, V) and Z (U, V) cannot be redefined as sections.

9 9.5.4 Simply contiguous array designators

- 10 1 A [section-subscript-list](#) specifies a [simply contiguous](#) section if and only if it does not have a [vector subscript](#)
11 and
- 12 • all but the last [subscript-triplet](#) is a colon,
 - 13 • the last [subscript-triplet](#) does not have a [stride](#), and
 - 14 • no [subscript-triplet](#) is preceded by a [section-subscript](#) that is a [subscript](#).
- 15 2 An array designator is [simply contiguous](#) if and only if it is
- 16 • 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

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)

X3D (:, i:j, 2)

Y5D (:, :, :, :, 7)

Passes part of the third column of ARRAY1.

Passes part of the second plane of X3D (or the whole plane if i==LBOUND (X3D, 2) and j==UBOUND (X3D, 2)).

Passes the seventh hypercube of Y5D.

All *simply contiguous* designators designate *contiguous* objects.

9.6 Image selectors

- 1 An image selector determines the *image index* for a *coindexed object*.

R926 *image-selector* is *lbracket cosubscript-list* [, *image-selector-spec-list*] *rbracket*

R927 *cosubscript* is *scalar-int-expr*

R928 *image-selector-spec* is NOTIFY = *notify-variable*

or STAT = *stat-variable*

or TEAM = *team-value*

or TEAM_NUMBER = *scalar-int-expr*

C932 No specifier shall appear more than once in a given *image-selector-spec-list*.

C933 A NOTIFY= *image-selector-spec* shall appear only in the designator of the variable of an *intrinsic assignment statement*.

C934 TEAM and TEAM_NUMBER shall not both appear in the same *image-selector-spec-list*.

C935 A *stat-variable* in an *image-selector* shall not be a *coindexed object*.

- 2 The number of *cosubscripts* shall be equal to the *corank* of the object. The value of a *cosubscript* in an image selector shall be within the *cobounds* for its *codimension*. Taking account of the *cobounds*, the *cosubscript* list in an image selector determines the *image index* in the same way that a subscript list in an array element determines the subscript order value (9.5.3.3), taking account of the bounds.

- 3 If a **TEAM= specifier** appears in an *image-selector*, the **team** of the image selector is specified by *team-value*, which shall identify the **current** or an **ancestor team**; the object shall be an **established coarray** in that **team**. If a **TEAM_NUMBER= specifier** appears in an *image-selector* and the current team is not the **initial team**, the value of the *scalar-int-expr* shall be equal to the value of a **team number** for a sibling team of the **current team** and the team of the image selector is that team; the object shall be an **established coarray** in the **parent** of the **current team**, or be an **associating entity** of the **CHANGE TEAM** construct. If a **TEAM_NUMBER= specifier** appears in an *image-selector* and the **current team** is the **initial team**, the value of *scalar-int-expr* shall be the **team number** for the **initial team**; the object shall be an **established coarray** in the **initial team**. Otherwise, the **team** of the image selector is the **current team**.
- 4 Execution of an assignment statement whose variable has a **NOTIFY= specifier** atomically increments the count of the corresponding notify variable on the image specified by the image selector, and does not wait for that image to execute a corresponding **NOTIFY WAIT statement**.
- 5 An image selector shall specify an **image index** value that is not greater than the number of **images** in the **team** of the image selector, and identifies the image with that index in that **team**.
- 6 Execution of a statement containing an *image-selector* with a **STAT= specifier** causes the *stat-variable* to become defined. If the designator is part of an operand that is evaluated or is a variable that is being defined or partly defined, and the object designated is on a **failed image**, the *stat-variable* is defined with the value **STAT_FAILED_IMAGE** (16.10.2.28) in the intrinsic module **ISO_FORTRAN_ENV**; otherwise, it is defined with the value zero.
- 7 The denotation of a *stat-variable* in an *image-selector* shall not depend on the evaluation of any entity in the same statement. The value of an expression shall not depend on the value of any *stat-variable* that appears in the same statement. The value of a *stat-variable* in an *image-selector* shall not be affected by the execution of any part of the statement, other than by whether the image specified by the *image-selector* has failed.

NOTE

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.

9.7 Dynamic association**9.7.1 ALLOCATE statement****9.7.1.1 Form of the ALLOCATE statement**

- 1 The **ALLOCATE** statement dynamically creates pointer **targets** and **allocatable** variables.
- | | | |
|------|------------------------|--|
| R929 | <i>allocate-stmt</i> | is ALLOCATE ([<i>type-spec</i> ::] <i>allocation-list</i> ■
■ [, <i>alloc-opt-list</i>]) |
| R930 | <i>alloc-opt</i> | is ERRMSG = <i>errmsg-variable</i>
or MOLD = <i>source-expr</i>
or SOURCE = <i>source-expr</i>
or STAT = <i>stat-variable</i> |
| R931 | <i>errmsg-variable</i> | is <i>scalar-default-char-variable</i> |

- 1 R932 *source-expr* is *expr*
- 2 R933 *allocation* is *allocate-object* [(*allocate-shape-spec-list*)] ■
 3 ■ [*lbracket allocate-coarray-spec rbracket*]
 4 or ([*lower-bounds-expr* :] *upper-bounds-expr*) ■
 5 ■ [*lbracket allocate-coarray-spec rbracket*]
- 6 R934 *allocate-object* is *variable-name*
 7 or *structure-component*
- 8 R935 *allocate-shape-spec* is [*lower-bound-expr* :] *upper-bound-expr*
- 9 R936 *lower-bound-expr* is *scalar-int-expr*
- 10 R937 *lower-bounds-expr* is *int-expr*
- 11 R938 *upper-bound-expr* is *scalar-int-expr*
- 12 R939 *upper-bounds-expr* is *int-expr*
- 13 R940 *allocate-coarray-spec* is [*allocate-coshape-spec-list* ,] [*lower-bound-expr* :] *
- 14 R941 *allocate-coshape-spec* is [*lower-bound-expr* :] *upper-bound-expr*
- 15 C936 (R934) Each *allocate-object* shall be a *data pointer* or an *allocatable* variable.
- 16 C937 (R929) 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.
- 17
 18 C938 (R929) If *type-spec* appears, it shall specify a type with which each *allocate-object* is *type compatible*.
 19
- 20 C939 (R929) A *type-param-value* in a *type-spec* shall be an asterisk if and only if each *allocate-object* is a
 21 *dummy argument* for which the corresponding type parameter is assumed.
- 22 C940 (R929) If *type-spec* appears, the kind type parameter values of each *allocate-object* shall be the
 23 same as the corresponding type parameter values of the *type-spec*.
- 24 C941 (R929) If an *allocate-object* is a *coarray*, *type-spec* shall not specify type *C_PTR* or type *C_FUNPTR*
 25 from the intrinsic module *ISO_C_BINDING*, or type *TEAM_TYPE* from the intrinsic module *ISO_-*
 26 *FORTRAN_ENV*.
- 27 C942 (R929) If an *allocate-object* is *unlimited polymorphic*, *type-spec* shall not specify a type that has a
 28 *coarray potential subobject component*.
- 29 C943 (R929) If an *allocate-object* is an array, either *allocate-shape-spec-list* or *upper-bounds-expr* shall
 30 appear in its *allocation*, or *source-expr* shall appear in the *ALLOCATE statement* and have the same
 31 *rank* as the *allocate-object*.
- 32 C944 (R933) If *allocate-object* is scalar, *allocate-shape-spec-list* shall not appear.
- 33 C945 (R933) An *allocate-coarray-spec* shall appear if and only if the *allocate-object* is a *coarray*.
- 34 C946 (R933) The number of *allocate-shape-specs* in an *allocate-shape-spec-list* shall be the same as the
 35 *rank* of the *allocate-object*. The number of *allocate-coshape-specs* in an *allocate-coarray-spec* shall
 36 be one less than the *corank* of the *allocate-object*.

- C947 If *upper-bounds-expr* and *lower-bounds-expr* both appear in an *allocation*, at least one of them shall be a rank-one array of constant size equal to the rank of *allocate-object*. Otherwise, if *upper-bounds-expr* appears in an *allocation*, it shall be a rank-one array of constant size equal to the rank of *allocate-object*.
- C948 (R930) No *alloc-opt* shall appear more than once in a given *alloc-opt-list*.
- C949 (R929) At most one of *source-expr* and *type-spec* shall appear.
- C950 (R929) Each *allocate-object* shall be *type compatible* (7.3.3) with *source-expr*. If *SOURCE=* appears, *source-expr* shall be a scalar or have the same *rank* as each *allocate-object*.
- C951 (R929) 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*.
- C952 (R929) The *declared type* of *source-expr* shall not be *C_PTR* or *C_FUNPTR* from the intrinsic module *ISO_C_BINDING*, or *TEAM_TYPE* from the intrinsic module *ISO_FORTRAN_ENV*, if an *allocate-object* is a *coarray*.
- C953 (R929) If an *allocate-object* is unlimited polymorphic, the *declared type* of *source-expr* shall not be a type that has a *coarray potential subobject component*.
- C954 (R929) If *SOURCE=* appears, the *declared type* of *source-expr* shall not be *EVENT_TYPE*, *LOCK_TYPE*, or *NOTIFY_TYPE* from the intrinsic module *ISO_FORTRAN_ENV*, or have a *potential subobject component* that is a *coarray* or of type *EVENT_TYPE*, *LOCK_TYPE*, or *NOTIFY_TYPE*.
- C955 (R934) An *allocate-object* shall not be a *coindexed object*.

NOTE 1

A pointer or allocatable component of a *coarray* can only be allocated by its own *image*.

```
TYPE (SOMETHING), ALLOCATABLE :: T[:]
```

```
...
```

```
ALLOCATE (T[*])
```

```
ALLOCATE (T%AAC (N))
```

```
ALLOCATE (T[Q]%AAC (N))
```

Allowed - implies synchronization.

Allowed - allocated by its own *image*.

Not allowed, because it is *coindexed*.

- 2 An *allocate-object* or a bound or type parameter of an *allocate-object* shall not depend on the value of *stat-variable*, the value of *errmsg-variable*, or on the value, bounds, *length type parameters*, allocation status, or association status of any *allocate-object* in the same *ALLOCATE* statement.
- 3 *source-expr* shall not be allocated within the *ALLOCATE* statement in which it appears; nor shall it depend on the value, *bounds*, *deferred type parameters*, allocation status, or association status of any *allocate-object* in that statement.
- 4 If an *ALLOCATE* statement has a *SOURCE=* specifier and an *allocate-object* that is a *coarray*, *source-expr* shall not have a *dynamic type* of *C_PTR* or *C_FUNPTR* from the intrinsic module *ISO_C_BINDING*, or *EVENT_TYPE*, *LOCK_TYPE*, *NOTIFY_TYPE*, or *TEAM_TYPE* from the intrinsic module *ISO_FORTRAN_ENV*, or have a *subcomponent* whose *dynamic type* is *EVENT_TYPE*, *LOCK_TYPE*, *NOTIFY_TYPE*, or *TEAM_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*. If an *allocate-object* is unlimited polymorphic, the *dynamic type* of *source-expr* shall not have a *coarray potential subobject component*.

- 6 If a *type-param-value* in a *type-spec* in an ALLOCATE statement is an asterisk, it denotes the current value of that assumed type parameter. If it is an expression, subsequent redefinition or undefinition of any entity in the expression does not affect the type parameter value.

NOTE 2

An example of an ALLOCATE statement is:

```
ALLOCATE (X (N), B (-3 : M, 0:9), STAT = IERR_ALLOC)
```

9.7.1.2 Execution of an ALLOCATE statement

- 1 When an ALLOCATE statement is executed for an array for which *allocate-shape-spec-list* is specified, the values of the lower bound and upper bound expressions determine the bounds of the array. Subsequent redefinition or undefinition of any entities in the bound expressions do not affect the array bounds. If the lower bound is omitted, the default value is one. 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 an array for which *upper-bounds-expr* is specified, it determines the upper bounds of the array. Subsequent redefinition or undefinition of an entity in a bounds expression does not affect the array bounds. If *lower-bounds-expr* appears, it determines the lower bounds; otherwise the default value is one. If *lower-bounds-expr* or *upper-bounds-expr* is scalar, the effect is as if it were broadcast to the shape of the other. If any element of *upper-bounds-expr* is less than the corresponding element of *lower-bounds-expr*, the extent in the corresponding dimension is zero and the array has zero size.
- 3 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*.
- 4 If an *allocation* specifies a *coarray*, its *dynamic type* and the values of corresponding type parameters shall be the same on every *active image* in the *current team*. The values of corresponding bounds and corresponding *cobounds* shall be the same on those *images*. The *coarray* shall be corresponding (5.4.7) on those *images*.
- 5 If an *allocation* specifies a *coarray*, the same ALLOCATE statement shall be executed on every *active image* of the *current team*. If the *coarray* is an *unsaved local variable* of a recursive procedure, the execution of the ALLOCATE statement shall be at the same depth of recursion of that procedure on those *images*.
- 6 When an ALLOCATE statement is executed for which an *allocate-object* is a *coarray*, there is an implicit synchronization of all *active images* in the *current team*. If the *current team* contains a *stopped* or *failed* image, an error condition occurs. If no other error condition occurs, execution on the *active images* of the *segment* (11.7.2) following the statement is delayed until all other *active images* in the *current team* have executed the same statement the same number of times in this team. The *segments* that executed before the ALLOCATE statement on an *active image* of this team precede the *segments* that execute after the ALLOCATE statement on another *active image* of this team. The *coarray* shall not become allocated on an *image* unless it is successfully allocated on all *active images* in this team.

NOTE

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

NOTE (cont.)

same on all [images](#) of the [current team](#), nor is it responsible for detecting or resolving deadlock problems (such as two [images](#) waiting on different [ALLOCATE statements](#)).

- 1 7 If [source-expr](#) is a pointer, it shall be associated with a [target](#). If [source-expr](#) is [allocatable](#), it shall be alloc-
2 ated.
- 3 8 When an ALLOCATE statement is executed for an array with no [allocate-shape-spec-list](#) or [upper-bounds-](#)
4 [expr](#), the array is allocated with the shape of [source-expr](#), and with each lower bound equal to the corres-
5 ponding element of [LBOUND \(source-expr\)](#). Subsequent changes to the bounds of [source-expr](#) do not affect
6 the array bounds.
- 7 9 If [SOURCE=](#) appears, [source-expr](#) shall be [conformable](#) with [allocation](#). If an [allocate-object](#) is not [poly-](#)
8 [morphic](#) and the [source-expr](#) is [polymorphic](#) with a [dynamic type](#) that differs from its [declared type](#), the
9 value provided for that [allocate-object](#) is the ancestor component of the [source-expr](#) that has the type of
10 the [allocate-object](#); otherwise the value provided is the value of the [source-expr](#). On successful allocation, if
11 [allocate-object](#) and [source-expr](#) have the same [rank](#) the value of [allocate-object](#) becomes the value provided,
12 otherwise the value of each element of [allocate-object](#) becomes the value provided. The [source-expr](#) is eval-
13 uated exactly once for each execution of an [ALLOCATE statement](#).
- 14 10 If [MOLD=](#) appears and [source-expr](#) is a variable, its value need not be defined.
- 15 11 If [type-spec](#) appears and the value of a [length type parameter](#) it specifies differs from the value of the corres-
16 ponding nondeferred type parameter specified in the declaration of any [allocate-object](#), an error condition
17 occurs. If the value of a nondeferred [length type parameter](#) of an [allocate-object](#) differs from the value of
18 the corresponding type parameter of [source-expr](#), an error condition occurs.
- 19 12 The set of error conditions for an ALLOCATE statement is processor dependent. If an error condition occurs
20 during execution of an ALLOCATE statement that does not contain the [STAT= specifier](#), [error termination](#)
21 is initiated. The [STAT= specifier](#) is described in 9.7.4. The [ERRMSG= specifier](#) is described in 9.7.5.

9.7.1.3 Allocation of allocatable variables

- 23 1 The allocation status of an [allocatable](#) entity is one of the following at any time.
 - 24 • The status of an [allocatable](#) variable becomes “allocated” if it is allocated by an ALLOCATE statement,
25 if it is allocated during assignment, or if it is given that status by the intrinsic subroutine [MOVE_-](#)
26 [ALLOC \(16.9.147\)](#). An [allocatable](#) variable with this status may be referenced, defined, or deallocated;
27 allocating it causes an error condition in the ALLOCATE statement. The result of the intrinsic function
28 [ALLOCATED \(16.9.13\)](#) is true for such a variable.
 - 29 • An [allocatable](#) variable has a status of “unallocated” if it is not allocated. The status of an [allocatable](#)
30 variable becomes unallocated if it is deallocated (9.7.3) or if it is given that status by the intrinsic
31 subroutine [MOVE_ALLOC](#). An [allocatable](#) variable with this status shall not be referenced or defined.
32 It shall not be supplied as an [actual argument](#) corresponding to a nonallocatable nonoptional [dummy](#)
33 [argument](#), except to certain intrinsic [inquiry functions](#). It may be allocated with the ALLOCATE state-
34 ment. Deallocating it causes an error condition in the DEALLOCATE statement. The result of the
35 intrinsic function [ALLOCATED \(16.9.13\)](#) is false for such a variable.
- 36 2 At the beginning of execution of a program, [allocatable](#) variables are unallocated.
- 37 3 When the allocation status of an [allocatable](#) variable changes, the allocation status of any associated [alloc-](#)
38 [atable](#) variable changes accordingly. Allocation of an [allocatable](#) variable establishes values for the [deferred](#)

1 **type parameters** of all associated **allocatable** variables.

2 4 An **unsaved allocatable local variable** of a procedure has a status of unallocated at the beginning of each
3 invocation of the procedure. An **unsaved allocatable local variable** of a construct has a status of unallocated
4 at the beginning of each execution of the construct.

5 5 When an object of derived type is created by an ALLOCATE statement, any **allocatable ultimate compon-**
6 **ents** have an allocation status of unallocated unless the **SOURCE= specifier** appears and the corresponding
7 component of the *source-expr* is allocated.

8 6 If the evaluation of a function would change the allocation status of a variable and if a reference to the func-
9 tion appears in an expression in which the value of the function is not needed to determine the value of the
10 expression, the allocation status of the variable after evaluation of the expression is processor dependent.

11 9.7.1.4 Allocation of pointer targets

12 1 Allocation of a **pointer** creates an object that implicitly has the **TARGET attribute**. Following successful exe-
13 cution of an ALLOCATE statement for a **pointer**, the **pointer** is associated with the **target** and can be used to
14 reference or define the **target**. Additional **pointers** can become associated with the **pointer target** or a part
15 of the **pointer target** by **pointer assignment**. It is not an error to allocate a **pointer** that is already associated
16 with a **target**. In this case, a new **pointer target** is created as required by the attributes of the **pointer** and
17 any array bounds, type, and type parameters specified by the ALLOCATE statement. The **pointer** is then
18 associated with this new **target**. Any previous association of the **pointer** with a **target** is broken. If the pre-
19 vious **target** had been created by allocation, it becomes inaccessible unless other **pointers** are associated
20 with it. The intrinsic function **ASSOCIATED** (16.9.20) can be used to determine whether a **pointer** that does
21 not have undefined association status is associated.

22 2 At the beginning of execution of a function whose result is a **pointer**, the association status of the result
23 **pointer** is undefined. Before such a function returns, it shall either associate a **target** with this **pointer** or
24 cause the association status of this **pointer** to become **disassociated**.

25 9.7.2 NULLIFY statement

26 R942 *nullify-stmt* **is** NULLIFY (*pointer-object-list*)

27 R943 *pointer-object* **is** *variable-name*
28 **or** *structure-component*
29 **or** *proc-pointer-name*

30 C956 (R943) Each *pointer-object* shall have the **POINTER attribute**.

31 1 A *pointer-object* shall not depend on the value, bounds, or association status of another *pointer-object* in
32 the same NULLIFY statement.

33 2 Execution of a NULLIFY statement causes each *pointer-object* to become **disassociated**.

NOTE

When a NULLIFY statement is applied to a **polymorphic pointer** (7.3.2.3), its **dynamic type** becomes the same as its **declared type**.

9.7.3 DEALLOCATE statement

9.7.3.1 Form of the DEALLOCATE statement

The DEALLOCATE statement causes [allocatable](#) variables to be deallocated; it causes pointer [targets](#) to be deallocated and the pointers to be [disassociated](#).

R944 *deallocate-stmt* **is** DEALLOCATE (*allocate-object-list* [, *dealloc-opt-list*])

R945 *dealloc-opt* **is** STAT = *stat-variable*
 or ERRMSG = *errmsg-variable*

C957 (R945) 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 9.7.4. The **ERRMSG= specifier** is described in 9.7.5.

4 When more than one allocated object is deallocated by execution of a DEALLOCATE statement, the order of deallocation is processor dependent.

NOTE

An example of a DEALLOCATE statement is:

```
DEALLOCATE (X, B)
```

9.7.3.2 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](#).

- 1 6 When a procedure is invoked, any allocated allocatable object that is an **actual argument** corresponding
 2 to an **INTENT (OUT) allocatable dummy argument** is deallocated; any allocated **allocatable** object that is a
 3 subobject of an **actual argument** corresponding to an **INTENT (OUT) dummy argument** is deallocated. If a
 4 Fortran procedure that has an **INTENT (OUT) allocatable dummy argument** is invoked by a C function and
 5 the corresponding argument in the C function call is a **C descriptor** that describes an allocated **allocatable**
 6 variable, the variable is deallocated on entry to the Fortran procedure. If a C function is invoked from a
 7 Fortran procedure via an interface with an **INTENT (OUT) allocatable dummy argument** and the corres-
 8 ponding **actual argument** in the reference to the C function is an allocated **allocatable** variable, the variable
 9 is deallocated on invocation (before execution of the C function begins).
- 10 7 When an **intrinsic assignment statement** (10.2.1.3) is executed, any noncoarray allocated **allocatable** sub-
 11 object of the variable is deallocated before the assignment takes place.
- 12 8 When a variable of derived type is deallocated, any allocated **allocatable** subobject is deallocated. If an error
 13 condition occurs during deallocation, it is **processor dependent** whether an allocated **allocatable subobject**
 14 is deallocated.
- 15 9 If an **allocatable** component is a subobject of a **finalizable** object, any final subroutine for that object is
 16 executed before the component is automatically deallocated.
- 17 10 When a statement that deallocates a **coarray** or an object with a **coarray potential subobject component** is
 18 executed, there is an implicit synchronization of all **active images** in the **current team**. If the **current team**
 19 contains a **stopped** or **failed** image, an error condition occurs. If no other error condition occurs, execution
 20 on the **active images** of the **segment** (11.7.2) following the statement is delayed until all other **active images**
 21 in the **current team** have executed the same statement the same number of times in this **team**. The **segments**
 22 that executed before the statement on an **active image** of this team precede the **segments** that execute after
 23 the statement on another **active image** of this team. A **coarray** shall not become deallocated on an **image**
 24 unless the corresponding **coarrays** are successfully deallocated on all other **active images** in this **team**.
- 25 11 If an **allocate-object** is a **coarray dummy argument**, the **ultimate arguments** (15.5.2.4) on those **images** shall
 26 be corresponding **coarrays**. If an **allocate-object** is a **coarray potential subobject component** of a **dummy**
 27 **argument**, those components of the **ultimate arguments** on those **images** shall be corresponding **coarrays**.
- 28 12 The effect of automatic deallocation is the same as that of a DEALLOCATE statement without a **dealloc-opt-**
 29 **list**.

NOTE 1

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 2

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 **active images** in the **current team**.

9.7.3.3 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.

9.7.4 STAT= specifier

R946 *stat-variable* is *scalar-int-variable*

- 1 A *stat-variable* should have a decimal exponent range of at least four; otherwise the processor-dependent error code might not be representable in the variable.
- 2 This rest of this subclause applies where an *alloc-opt* or *dealloc-opt* that is a STAT= specifier appears in an ALLOCATE or DEALLOCATE statement.
- 3 The *stat-variable* shall not be allocated or deallocated within the ALLOCATE or DEALLOCATE statement in which it appears; nor shall it depend on the value, *bounds*, *deferred type parameters*, allocation status, or association status of any *allocate-object* in that statement. The *stat-variable* shall not depend on the value of the *errmsg-variable*.
- 4 Successful execution of the ALLOCATE or DEALLOCATE statement causes the *stat-variable* to become defined with a value of zero.
- 5 If an ALLOCATE statement with a *coarray allocate-object*, or a DEALLOCATE statement with an *allocate-object* that is a *coarray* or which has a *coarray potential subobject component*, is executed when the *current team* contains a *stopped image*, the *stat-variable* becomes defined with the value STAT_STOPPED_IMAGE from the intrinsic module ISO_FORTRAN_ENV (16.10.2). Otherwise, if such a statement is executed when the *current team* contains a *failed image*, and no other error condition occurs, the *stat-variable* becomes defined with value STAT_FAILED_IMAGE from the intrinsic module ISO_FORTRAN_ENV. If any other error condition occurs during execution of the ALLOCATE or DEALLOCATE statement, the *stat-variable* becomes defined with a processor-dependent positive integer value different from STAT_STOPPED_IMAGE and STAT_FAILED_IMAGE.
- 6 If *stat-variable* became defined with the value STAT_FAILED_IMAGE, each *allocate-object* is successfully allocated or deallocated on all the *active images* of the *current team*. If any other error condition occurs, 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

The status of objects that were not successfully allocated or deallocated can be individually checked with the intrinsic functions [ALLOCATED](#) or [ASSOCIATED](#).

9.7.5 ERRMSG= specifier

- 1
 - 2
 - 3
 - 4
 - 5
 - 6
 - 7
 - 8
- 1 The *errmsg-variable* shall not be an *allocate-object* of the [ALLOCATE](#) or [DEALLOCATE](#) statement in which it appears; nor shall it depend on the value, [bounds](#), [deferred type parameters](#), allocation status, or association status of any *allocate-object* in that statement. The *errmsg-variable* shall not depend on the value of the *stat-variable*.
 - 2 If an error condition occurs during execution of an [ALLOCATE](#) or [DEALLOCATE](#) statement with an [ERRMSG=](#) specifier, the *errmsg-variable* is assigned an explanatory message, as if by [intrinsic assignment](#). If no such condition occurs, the definition status and value of *errmsg-variable* are unchanged.

10 Expressions and assignment

10.1 Expressions

10.1.1 Expression semantics

- 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 (10.1.9). The *corank* of an expression that is not a variable is zero.

10.1.2 Form of an expression

10.1.2.1 Overall expression syntax

- 1 An expression is formed from operands, operators, and parentheses. An operand is either a scalar or an array. An operation is either intrinsic (10.1.5) or defined (10.1.6). More complicated expressions can be formed using operands which are themselves expressions.
- 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*.

10.1.2.2 Primary

R1001 *primary* is *literal-constant*
 or *designator*
 or *array-constructor*
 or *structure-constructor*
 or *enum-constructor*
 or *enumeration-constructor*
 or *function-reference*
 or *type-param-inquiry*
 or *type-param-name*
 or (*expr*)
 or *conditional-expr*

C1001 (R1001) The *type-param-name* shall be the name of a type parameter.

C1002 (R1001) The *designator* shall not be a whole *assumed-size array*.

C1003 (R1001) The *expr* shall not be a function reference that returns a *procedure pointer*.

NOTE

Examples of a *primary* are:

Example	Syntactic class
1.0	<i>constant</i>

NOTE (cont.)

'ABCDEFGHIJKLMNOPQRSTUVWXYZ' (I:I)	<i>designator</i>
[1.0, 2.0]	<i>array-constructor</i>
PERSON ('Jones', 12)	<i>structure-constructor</i>
F (X, Y)	<i>function-reference</i>
X%KIND	<i>type-param-inquiry</i>
KIND	<i>type-param-name</i>
(S + T)	<i>(expr)</i>

10.1.2.3 Conditional expressions

1 A conditional expression is a primary that selectively evaluates a chosen subexpression.

R1002 *conditional-expr* **is** [*scalar-logical-expr* ? *expr* [: *scalar-logical-expr* ? *expr*]... : *expr*]

C1004 Each *expr* of a *conditional-expr* shall have the same **declared type**, **kind type parameters**, and rank.

NOTE

Examples of a conditional expression are:

```
( ABS (RESIDUAL)<=TOLERANCE ? 'ok' : 'did not converge' )
( I>0 .AND. I<=SIZE (A) ? A (I) : PRESENT (VAL) ? VAL : 0.0 )
```

10.1.2.4 Level-1 expressions

1 Defined unary operators have the highest operator precedence (Table 10.1). A level-1 expression is a primary optionally operated on by a defined unary operator:

R1003 *level-1-expr* **is** [*defined-unary-op*] *primary*

R1004 *defined-unary-op* **is** . *letter* [*letter*]

C1005 (R1004) 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

Simple examples of a level-1 expression are:

<u>Example</u>	<u>Syntactic class</u>
A	<i>primary</i> (R1001)
.INVERSE. B	<i>level-1-expr</i> (R1003)

A more complicated example of a level-1 expression is:

```
.INVERSE. (A + B)
```

10.1.2.5 Level-2 expressions

1 Level-2 expressions are level-1 expressions optionally involving the numeric operators *power-op*, *mult-op*, and *add-op*.

R1005 *mult-operand* **is** *level-1-expr* [*power-op mult-operand*]

R1006 *add-operand* **is** [*add-operand mult-op*] *mult-operand*

R1007 *level-2-expr* **is** [[*level-2-expr*] *add-op*] *add-operand*

1	R1008 <i>power-op</i>	is **
2	R1009 <i>mult-op</i>	is *
3		or /
4	R1010 <i>add-op</i>	is +
5		or -

NOTE

Simple examples of a level-2 expression are:

<u>Example</u>	<u>Syntactic class</u>	<u>Remarks</u>
A	<i>level-1-expr</i>	A is a <i>primary</i> . (R1003)
B ** C	<i>mult-operand</i>	B is a <i>level-1-expr</i> , ** is a <i>power-op</i> , and C is a <i>mult-operand</i> . (R1005)
D * E	<i>add-operand</i>	D is an <i>add-operand</i> , * is a <i>mult-op</i> , and E is a <i>mult-operand</i> . (R1006)
+1	<i>level-2-expr</i>	+ is an <i>add-op</i> and 1 is an <i>add-operand</i> . (R1007)
F - I	<i>level-2-expr</i>	F is a <i>level-2-expr</i> , - is an <i>add-op</i> , and I is an <i>add-operand</i> . (R1007)

A more complicated example of a level-2 expression is:

- A + D * E + B ** C

6 10.1.2.6 Level-3 expressions

7 1 Level-3 expressions are level-2 expressions optionally involving the character operator *concat-op*.

8	R1011 <i>level-3-expr</i>	is [<i>level-3-expr concat-op</i>] <i>level-2-expr</i>
9	R1012 <i>concat-op</i>	is //

NOTE

Simple examples of a level-3 expression are:

<u>Example</u>	<u>Syntactic class</u>
A	<i>level-2-expr</i> (R1007)
B // C	<i>level-3-expr</i> (R1011)

A more complicated example of a level-3 expression is:

X // Y // 'ABCD'

10 10.1.2.7 Level-4 expressions

11 1 Level-4 expressions are level-3 expressions optionally involving the relational operators *rel-op*.

12	R1013 <i>level-4-expr</i>	is [<i>level-3-expr rel-op</i>] <i>level-3-expr</i>
13	R1014 <i>rel-op</i>	is .EQ.
14		or .NE.
15		or .LT.
16		or .LE.
17		or .GT.
18		or .GE.
19		or ==
20		or /=

1	or	<
2	or	<=
3	or	>
4	or	>=

NOTE

Simple examples of a level-4 expression are:

Example	Syntactic class
A	<i>level-3-expr</i> (R1011)
B == C	<i>level-4-expr</i> (R1013)
D < E	<i>level-4-expr</i> (R1013)

A more complicated example of a level-4 expression is:

$$(A + B) \neq C$$

10.1.2.8 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*.

R1015 *and-operand* is *[not-op] level-4-expr*

R1016 *or-operand* is [*or-operand and-op*] *and-operand*

R1017 *equiv-operand* **is** [*equiv-operand or-op*] *or-operand*

R1018 *level-5-expr* is $[\textit{level-5-expr equiv-op}] \textit{equiv-operand}$

R1019 *not-op* is **.NOT.**

R1020 *and-op* is **.AND.**

R1021 *or-op* is **.OR.**

R1022 *equiv-op* **is** .EQV.
 or .NEQV.

NOTE

Simple examples of a level-5 expression are:

Example	Syntactic class
A	<i>level-4-expr</i> (R1013)
.NOT. B	<i>and-operand</i> (R1015)
C .AND. D	<i>or-operand</i> (R1016)
E .OR. F	<i>equiv-operand</i> (R1017)
G .EQV. H	<i>level-5-expr</i> (R1018)
S .NEQV. T	<i>level-5-expr</i> (R1018)

A more complicated example of a level-5 expression is:

A .AND. B .EQV. .NOT. C

10.1.2.9 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 10.1).

- 1 R1023 *expr* is [*expr defined-binary-op*] *level-5-expr*
- 2 R1024 *defined-binary-op* is . *letter* [*letter*]
- 3 C1006 (R1024) A *defined-binary-op* shall not contain more than 63 letters and shall not be the same as
- 4 any *intrinsic-operator* or *logical-literal-constant*.

NOTE

Simple examples of an expression are:

Example	Syntactic class
A	<i>level-5-expr</i> (R1018)
B.UNION.C	<i>expr</i> (R1023)

More complicated examples of an expression are:

```
(B .INTERSECT. C) .UNION. (X - Y)
A + B == C * D
.INVERSE. (A + B)
A + B .AND. C * D
E // G == H (1:10)
```

5 10.1.3 Precedence of operators

- 6 1 There is a precedence among the intrinsic and extension operations corresponding to the form of expres-
- 7 sions specified in 10.1.2, which determines the order in which the operands are combined unless the order
- 8 is changed by the use of parentheses. This precedence order is summarized in Table 10.1.

Table 10.1 — Categories of operations and relative precedence

Category of operation	Operators	Precedence
Extension	<i>defined-unary-op</i>	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	<i>defined-binary-op</i>	Lowest

- 9 2 The precedence of a *defined operation* is that of its operator.

NOTE 1

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)
```

- 10 3 The general form of an expression (10.1.2) also establishes a precedence among operators in the same
- 11 syntactic class. This precedence determines the order in which the operands are to be combined in de-
- 12 termining the interpretation of the expression unless the order is changed by the use of parentheses.

NOTE 2

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

```
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 (10.1.2), only the first *add-operand* of a *level-2-expr* can be preceded by the identity (+) or negation (−) operator. These formation rules do not permit expressions containing two consecutive numeric operators, such as A ** −B or A + −B. However, expressions such as A ** (−B) and A + (−B) are permitted. The rules do allow a binary operator or an intrinsic unary operator to be followed by a defined unary operator, such as:

```
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 3

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 4

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.

1 10.1.4 Evaluation of operations

- 1 An intrinsic operation requires the values of its operands.
- 2 Execution of a function reference in the logical expression in an *IF statement* (11.1.8.4), the mask expression

in a [WHERE statement \(10.2.3.1\)](#), or the *concurrent-limits* and *concurrent-steps* in a [FORALL statement¹ \(10.2.4\)](#) is permitted to define variables in the subsidiary *action-stmt*, *where-assignment-stmt*, or *forall-assignment-stmt* respectively. Except in those cases:

- the evaluation of a function reference shall neither affect nor be affected by the evaluation of any other entity within the statement;
- if a function reference causes definition or undefinition of an [actual argument](#) of the function, that argument or any associated entities shall not appear elsewhere in the same statement.

NOTE 1

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 contains.

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 2

For example, the array expression

```
A + B
```

produces an array of the same shape as A and B. The individual array elements of the result have the values of the first element of A added to the first element of B, the second element of A added to the second element of B, etc.

5 When an [elemental](#) unary operator operates on an array operand, the operation is performed element-by-element, and the result is the same shape as the operand. If an [elemental operation](#) is intrinsically pure or is implemented by a pure [elemental](#) function (15.9), the element operations may be performed simultaneously or in any order.

6 Evaluation of a *conditional-expr* evaluates each *scalar-logical-expr* in order, until the value of a *scalar-logical-expr* is true, or there are no more *scalar-logical-exprs*. If the value of a *scalar-logical-expr* is true, its subsequent *expr* is chosen; otherwise, the last *expr* of the *conditional-expr* is chosen. The chosen *expr* is evaluated, and its value is the value of the conditional expression.

7 The declared type, kind type parameters, and rank of a *conditional-expr* are the same as those of its *exprs*. The dynamic type, length type parameters, and shape are those of the chosen *expr*. A *conditional-expr* is polymorphic if and only if one or more of its *exprs* is polymorphic.

NOTE 3

Only one *expr* of a conditional expression is evaluated, and any of its *scalar-logical-exprs* subsequent to one that evaluates to true are not evaluated.

¹The [FORALL statement](#) is obsolescent.

10.1.5 Intrinsic operations

10.1.5.1 Intrinsic operation classification

1 An intrinsic operation is either a unary or binary operation. An intrinsic unary operation is an operation of the form *intrinsic-operator* x_2 where x_2 is of a type (7.4, 7.6) listed in Table 10.2 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 types listed in Table 10.2 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. An enumeration relational intrinsic operation is a relational intrinsic operation for which both operands are of the same enumeration type. An enum relational intrinsic operation is a relational intrinsic operation for which one operand is of an enum type, and the other operand has the same type or is an integer expression involving an enumerator of that type. The **kind type parameters** of the operands of a character relational intrinsic operation shall be the same.

7 The interpretations defined in 10.1.5 apply to both scalars and arrays; the interpretation for arrays is obtained by applying the interpretation for scalars element by element.

Table 10.2 — Types of operands and results for intrinsic operators

Intrinsic operator <i>op</i>	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, N	L, L, L, L
	R	I, R, Z	L, L, L
	Z	I, R, Z	L, L, L
	C	C	L
	E	E	L
	N	N, I	L, L
.GT., .GE., .LT., .LE. >, >=, <, <=	I	I, R, N	L, L, L
	R	I, R	L, L
	C	C	L
	E	E	L
	N	N, I	L, L
.NOT.		L	L
.AND., .OR., .EQV., .NEQV.	L	L	L

Types of operands and results for intrinsic operators (cont.)

Intrinsic operator op	Type of x_1	Type of x_2	Type of $[x_1] op x_2$
The symbols I, R, Z, C, and L stand for the types integer, real, complex, character, and logical, respectively. The symbol E stands for the same enumeration type for both operands. The symbol N stands for an enum type, where if the other operand is N, they have the same type, and if the other operand is I, the integer operand is an expression with a primary that is an enumerator of the enum type. 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.			

NOTE

For example, if X is of type real and J is of type integer, the expression X + J is of type real.

10.1.5.2 Numeric intrinsic operations**10.1.5.2.1 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 of type real or complex 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 10.1.5.1.

3 The numeric operators and their interpretation in an expression are given in Table 10.3, where x_1 denotes the operand to the left of the operator and x_2 denotes the operand to the right of the operator.

Table 10.3 — 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 (10.1.5.2.2).

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 $1/(x_1 ** \text{ABS}(x_2))$, which is subject to the rules of integer division (10.1.5.2.2).

NOTE

For example, $2 ** (-3)$ has the value of $1/(2 ** 3)$, which is zero.

10.1.5.2.2 Integer division

One operand of type integer may be divided by another operand of type integer. Although the mathematical quotient of two integers is not necessarily an integer, Table 10.2 specifies that an expression involving the division operator with two operands of type integer is interpreted as an expression of type integer. The result of such an operation is the integer closest to the mathematical quotient and between zero and the mathematical quotient inclusively.

NOTE

For example, the expression $(-8) / 3$ has the value (-2) .

10.1.5.2.3 Complex exponentiation

In the case of a complex value raised to a complex power, the value of the operation $x_1^{**} x_2$ is the principal value of $x_1^{x_2}$.

10.1.5.2.4 Evaluation of numeric intrinsic operations

- The execution of any numeric operation whose result is not defined by the arithmetic used by the processor is prohibited. Raising a negative real value to a real power is prohibited.
- Once the interpretation of a numeric intrinsic operation is established, the processor may evaluate any mathematically equivalent expression, provided that the integrity of parentheses is not violated.
- Two expressions of a numeric type are mathematically equivalent if, for all possible values of their primaries, their mathematical values are equal. However, mathematically equivalent expressions of numeric type can produce different computational results.

NOTE 1

Any difference between the values of the expressions $(1./3.)*3.$ and $1.$ is a computational difference, not a mathematical difference. The difference between the values of the expressions $5/2$ and $5./2.$ is a mathematical difference, not a computational difference.

The mathematical definition of integer division is given in 10.1.5.2.2.

NOTE 2

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
$\overline{X + Y}$	$Y + X$
$X * Y$	$Y * X$
$-X + Y$	$Y - X$
$X + Y + Z$	$X + (Y + Z)$
$X - Y + Z$	$X - (Y - Z)$
$X * A / Z$	$X * (A / Z)$
$X * Y - X * Z$	$X * (Y - Z)$
$A / B / C$	$A / (B * C)$
$A / 5.0$	$0.2 * A$

The following are examples of expressions with forbidden alternative forms that cannot be used by a processor in the evaluation of those expressions.

NOTE 2 (cont.)

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 3

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 4

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.

1 **10.1.5.3 Character intrinsic operation**

2 **10.1.5.3.1 Interpretation of the character intrinsic operation**

- 3 1 The character intrinsic operator `//` is used to concatenate two operands of type character with the same
- 4 [kind type parameter](#). Evaluation of the character intrinsic operation produces a result of type character.
- 5 2 The interpretation of the character intrinsic operator `//` when used to form an expression is given in Table
- 6 [10.4](#), where x_1 denotes the operand to the left of the operator and x_2 denotes the operand to the right of
- 7 the operator.

Table 10.4 — Interpretation of the character intrinsic operator `//`

Operator	Representing	Use of operator	Interpretation
<code>//</code>	Concatenation	$x_1 // x_2$	Concatenate x_1 with x_2

- 8 3 The result of the character intrinsic operation $x_1 // x_2$ is a character string whose value is the value of x_1
- 9 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 .
- 10 Parentheses used to specify the order of evaluation have no effect on the value of a character expression.

NOTE

For example, the value of the expression `(AB' // 'CDE') // 'F'` is the string 'ABCDEF'. The value of the expression `AB' // ('CDE' // 'F')` is also the string 'ABCDEF'.

10.1.5.3.2 Evaluation of the character intrinsic operation

1 A processor is only required to evaluate as much of the character intrinsic operation as is required by the context in which the expression appears.

NOTE

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.

10.1.5.4 Logical intrinsic operations**10.1.5.4.1 Interpretation of logical intrinsic operations**

1 A logical operation is used to express a logical computation. Evaluation of a logical operation produces a result of type logical. The permitted types for operands of the logical intrinsic operations are specified in [10.1.5.1](#).

2 The logical operators and their interpretation when used to form an expression are given in Table [10.5](#), where x_1 denotes the operand to the left of the operator and x_2 denotes the operand to the right of the operator.

Table 10.5 — 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 [10.6](#).

Table 10.6 — 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

10.1.5.4.2 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

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)

- 1 2 Two expressions of type logical are logically equivalent if their values are equal for all possible values of
2 their primaries.

3 **10.1.5.5 Relational intrinsic operations**

4 **10.1.5.5.1 Interpretation of relational intrinsic operations**

- 5 1 A relational intrinsic operation is used to compare values of two operands using the relational intrinsic
6 operators .LT., .LE., .GT., .GE., .EQ., .NE., <, <=, >, >=, ==, and /=. The permitted types for operands of the
7 relational intrinsic operators are specified in 10.1.5.1.

- 8 2 The operators <, <=, >, >=, ==, and /= always have the same interpretations as the operators .LT., .LE., .GT.,
9 .GE., .EQ., and .NE., respectively.

NOTE 1

As shown in Table 10.2, a relational intrinsic operator cannot be used to compare the value of an expression of a numeric type with one of type character or logical. Also, two operands of type logical cannot be 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.

- 10 3 Evaluation of a relational intrinsic operation produces a default logical result.
- 11 4 The interpretation of the relational intrinsic operators is given in Table 10.7, where x_1 denotes the operand
12 to the left of the operator and x_2 denotes the operand to the right of the operator.

Table 10.7 — 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

- 13 5 A numeric relational intrinsic operation is interpreted as having the logical value true if and only if the
14 values of the operands satisfy the relation specified by the operator.

- 15 6 In the numeric relational operation

16 x_1 *rel-op* x_2

- 17 if the types or kind type parameters of x_1 and x_2 differ, their values are converted to the type and kind type
18 parameter of the expression $x_1 + x_2$ before evaluation.

- 1 7 A character relational intrinsic operation is interpreted as having the logical value true if and only if the
2 values of the operands satisfy the relation specified by the operator.
- 3 8 For a character relational intrinsic operation, the operands are compared one character at a time in order,
4 beginning with the first character of each character operand. If the operands are of unequal length, the
5 shorter operand is treated as if it were extended on the right with blanks to the length of the longer operand.
6 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
7 in the corresponding position in x_2 , x_1 is equal to x_2 . Otherwise, at the first position where the character
8 operands differ, the character operand x_1 is considered to be less than x_2 if the character value of x_1 at
9 this position precedes the value of x_2 in the [collating sequence](#) (3.27); x_1 is greater than x_2 if the character
10 value of x_1 at this position follows the value of x_2 in the [collating sequence](#).

NOTE 2

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.

- 11 9 An enumeration relational intrinsic operation is interpreted as having the logical value true if and only if
12 the ordinal values of the operands satisfy the relation specified by the operator.
- 13 10 An enum relational intrinsic operation is interpreted as if all operands of enum type were converted to
14 their corresponding integer values.

15 10.1.5.2 Evaluation of relational intrinsic operations

- 16 1 Once the interpretation of a relational intrinsic operation is established, the processor may evaluate any
17 other expression that is relationally equivalent, provided that the integrity of parentheses in any expression
18 is not violated.
- 19 2 Two relational intrinsic operations are relationally equivalent if their logical values are equal for all possible
20 values of their primaries.

NOTE

Whether an operand of a relational intrinsic operation could be an [IEEE NaN](#) affects whether expressions are equivalent. For example, if x or y could be a [NaN](#), the expressions

`.NOT. (x .LT. y)` and `x .GE. y`

are not equivalent.

21 10.1.6 Defined operations

22 10.1.6.1 Definitions

- 23 1 A [defined operation](#) is either a unary operation or a binary operation. A unary [defined operation](#) is an
24 operation that has the form [defined-unary-op](#) x_2 or [intrinsic-operator](#) x_2 and that is defined by a function
25 and a [generic interface](#) (7.5.5, 15.4.3.4).
- 26 2 A function defines the unary operation op x_2 if
- 27 (1) the function is specified with a [FUNCTION](#) (15.6.2.2) or [ENTRY](#)² (15.6.2.6) statement that spe-
28 cifies one [dummy argument](#) d_2 ,

²The [ENTRY statement](#) is obsolescent.

- (2) either
 - (a) a [generic interface](#) (15.4.3.2) provides the function with a *generic-spec* of **OPERATOR** (*op*), or
 - (b) there is a generic [binding](#) (7.5.5) 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) d_2 is type-compatible with 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 with 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** (15.6.2.2) or **ENTRY**³ (15.6.2.6) statement that specifies two [dummy arguments](#), d_1 and d_2 ,
- (2) either
 - (a) a [generic interface](#) (15.4.3.2) provides the function with a *generic-spec* of **OPERATOR** (*op*), or
 - (b) there is a generic [binding](#) (7.5.5) 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) d_1 and d_2 are type-compatible with 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 , respectively, 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

An intrinsic operator can be used as the operator in a [defined operation](#). In such a case, the generic properties of the operator are extended.

10.1.6.2 Interpretation of a defined operation

- 1 The interpretation of a [defined operation](#) is provided by the function that defines the operation.
- 2 The operators $<$, $<=$, $>$, $>=$, $==$, and $\neq$ always have the same interpretations as the operators [.LT.](#), [.LE.](#), [.GT.](#), [.GE.](#), [.EQ.](#), and [.NE.](#), respectively.

³The [ENTRY statement](#) is obsolescent.

10.1.6.3 Evaluation of a defined operation

- 1 Once the interpretation of a **defined operation** is established, the processor may evaluate any other expression that is equivalent, provided that the integrity of parentheses is not violated.
- 2 Two expressions of derived type are equivalent if their values are equal for all possible values of their primaries.

10.1.7 Evaluation of operands

- 1 It is not necessary for a processor to evaluate all of the operands of an expression, or to evaluate entirely each operand, if the value of the expression can be determined otherwise.

NOTE 1

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

$$X > Y \text{ .OR. } L(Z)$$

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 2

In the examples in NOTE 1, 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 3

This restriction is intended to avoid inadvertent deadlock caused by optimization.

10.1.8 Integrity of parentheses

- 1 The rules for evaluation specified in 10.1.5 state certain conditions under which a processor can evaluate an expression that is different from the one specified by applying the rules given in 10.1.2 and the rules for interpretation specified in 10.1.5. However, any expression in parentheses shall be treated as a data entity.

NOTE

For example, in evaluating the expression $A + (B - C)$ where A, B, and C are of **numeric types**, the difference of B and C shall be evaluated before the addition operation is performed; the processor shall not evaluate the mathematically equivalent expression $(A + B) - C$.

10.1.9 Type, type parameters, and shape of an expression

10.1.9.1 General

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 (7.4) or a nonintrinsic type (7.5, 7.6).

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.

R1025 *logical-expr* is *expr*

C1007 (R1025) *logical-expr* shall be of type logical.

R1026 *default-char-expr* is *expr*

C1008 (R1026) *default-char-expr* shall be default character.

R1027 *int-expr* is *expr*

C1009 (R1027) *int-expr* shall be of type integer.

R1028 *numeric-expr* is *expr*

C1010 (R1028) *numeric-expr* shall be of type integer, real, or complex.

10.1.9.2 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 [literal constant](#), [designator](#), array constructor, [structure constructor](#), enum constructor, enumeration constructor, function reference, [type parameter inquiry](#), type parameter name, or parenthesized expression. If a primary is a [literal constant](#), its type, [type parameters](#), and shape are those of the [literal constant](#). If it is a [structure constructor](#), it is scalar and its type and [type parameters](#) are as described in 7.5.10. If it is an enum constructor, it is scalar and its type is as described in 7.6.1. If it is an enumeration constructor, it is scalar and its type is as described in 7.6.2. If it is an array constructor, its type, [type parameters](#), and shape are as described in 7.8. If it is a [designator](#) or function reference, its type, [type parameters](#), and shape are those of the [designator](#) (8.2, 8.5) or the function reference (15.5.3), respectively. If the function reference is generic (15.4.3.2, 16.7) then its type, [type parameters](#), and shape are those of the specific function referenced, which is determined by the declared types, [type parameters](#), and [ranks](#) of its [actual arguments](#) as specified in 15.5.5.2. If it is a [type parameter inquiry](#) or type parameter name, it is a scalar integer with the kind of the type parameter.

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](#). The type, type parameters, and shape of the primary are those of the [target](#). If the pointer is not associated with a [target](#), it shall appear as a primary only as an [actual argument](#) in a reference to a procedure whose corresponding [dummy argument](#) is declared to be a pointer, as the target in a [pointer](#)

assignment statement, or as explicitly permitted elsewhere in this document.

A disassociated array pointer or an unallocated allocatable array has no shape but does have rank. The type, type parameters, and rank of the result of the intrinsic function NULL (16.9.155) depend on context.

10.1.9.3 Type, type parameters, and shape of the result of an operation

The type of the result of an intrinsic operation $[x_1] \text{ op } x_2$ is specified by Table 10.2. The shape of the result of an intrinsic operation is the shape of x_2 if op is unary or if x_1 is scalar, and is the shape of x_1 otherwise.

The type, type parameters, and shape of the result of a defined operation $[x_1] \text{ op } x_2$ are specified by the function defining the operation (10.1.6).

An expression of an intrinsic type has a kind type parameter. An expression of type character also has a character length parameter.

The type parameters of the result of an intrinsic operation are as follows.

- For an expression $x_1 // x_2$ where $//$ is the character intrinsic operator and x_1 and x_2 are of type character, the character length parameter is the sum of the lengths of the operands and the kind type parameter is the kind type parameter of x_1 , which shall be the same as the kind type parameter of x_2 .
- For an expression $\text{op } x_2$ where op is an intrinsic unary operator and x_2 is of type integer, real, complex, or logical, the kind type parameter of the expression is that of the operand.
- For an expression $x_1 \text{ op } x_2$ where op is a numeric intrinsic binary operator with 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 $x_1 \text{ op } x_2$ 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 $x_1 \text{ op } x_2$ where op is a logical intrinsic binary operator with both operands of the same kind type parameter, the kind type parameter of the expression is identical to that of each operand. In the case where both operands are of type logical with different kind type parameters, the kind type parameter of the expression is processor dependent, but it is the same as that of one of the operands.
- For an expression $x_1 \text{ op } x_2$ where op is a relational intrinsic operator, the kind type parameter of the expression is default logical.

10.1.10 Conformability rules for elemental operations

An elemental operation is an intrinsic operation or a defined operation for which the function is elemental (15.9).

- 1 2 For all **elemental** binary operations, the two operands shall be conformable. In the case where one is a
 2 scalar and the other an array, the scalar is treated as if it were an array of the same shape as the array
 3 operand with every element, if any, of the array equal to the value of the scalar.

4 **10.1.11 Specification expression**

- 5 1 A **specification expression** is an expression with limitations that make it suitable for use in specifications
 6 such as length type parameters (C704) and array bounds (R816, R817). A *specification-expr* shall be a **con-**
 7 **stant expression** unless it is in an **interface body** (15.4.3.2), the specification part of a subprogram or **BLOCK**
 8 **construct**, a derived type definition, or the *declaration-type-spec* of a **FUNCTION statement** (15.6.2.2).

9 R1029 *specification-expr* **is** *scalar-int-expr*

10 C1011 (R1029) The *scalar-int-expr* shall be a restricted expression.

- 11 2 A restricted expression is an expression in which each operation is intrinsic or defined by a specification
 12 function and each primary is

- 13 (1) a constant or subobject of a constant,
- 14 (2) an **object designator** with a **base object** that is a **dummy argument** that has neither the **OP-**
 15 **TIONAL** nor the **INTENT (OUT)** attribute,
- 16 (3) an **object designator** with a **base object** that is in a **common block**⁴,
- 17 (4) an **object designator** with a **base object** that is made accessible by **use** or **host** association,
- 18 (5) an array constructor where each element and each *scalar-int-expr* of each *ac-implied-do-control*
 19 is a restricted expression,
- 20 (6) a **structure constructor** where each component is a restricted expression,
- 21 (7) an enum constructor whose *expr* is a restricted expression,
- 22 (8) an enumeration constructor whose *expr* is a restricted expression,
- 23 (9) a specification inquiry where each **designator** or argument is
 - 24 (a) a restricted expression or
 - 25 (b) a variable that is not an optional dummy argument, and whose properties inquired about
 26 are not
 - 27 (i) dependent on the upper bound of the last dimension of an **assumed-size array**,
 - 28 (ii) deferred, or
 - 29 (iii) defined by an expression that is not a restricted expression,
- 30 (10) a specification inquiry that is a constant expression,
- 31 (11) a reference to the intrinsic function **PRESENT**,
- 32 (12) a reference to any other **standard intrinsic** function where each argument is a restricted expres-
 33 sion,
- 34 (13) a reference to a transformational function from the intrinsic module **IEEE_ARITHMETIC**, **IEEE_-**
 35 **EXCEPTIONS**, or **ISO_C_BINDING**, where each argument is a restricted expression,
- 36 (14) a reference to a specification function where each argument is a restricted expression,
- 37 (15) a type parameter of the derived type being defined,
- 38 (16) an *ac-do-variable* within an array constructor where each *scalar-int-expr* of the corresponding
 39 *ac-implied-do-control* is a restricted expression, or
- 40 (17) a restricted expression enclosed in parentheses,

⁴Common blocks are obsolescent.

where each subscript, section subscript, substring starting point, substring ending point, and type parameter value is a restricted expression.

3 A specification inquiry is a reference to

- (1) an intrinsic **inquiry function** other than **PRESENT**,
- (2) a **type parameter inquiry** (9.4.5),
- (3) an **inquiry function** from the intrinsic module **IEEE_ARITHMETIC** or the intrinsic module **IEEE_EXCEPTIONS** (17.10),
- (4) the function **C_SIZEOF** from the intrinsic module **ISO_C_BINDING** (18.2.3.8), or
- (5) the **COMPILER_VERSION** or **COMPILER_OPTIONS** function from the intrinsic module **ISO_FORTRAN_ENV** (16.10.2.6, 16.10.2.7).

4 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 function⁵, and does not have a **dummy procedure** argument.

5 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 1

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.

6 A variable in a **specification expression** shall have its type and type parameters, if any, specified by a previous declaration in the same **scoping unit**, by the implicit typing rules in effect for the **scoping unit**, or by **host** or **use** association. If a variable in a **specification expression** is typed by the implicit typing rules, its appearance in any subsequent **type declaration statement** shall confirm the implied type and type parameters. If a specification inquiry depends on the type of an object of derived type, that type shall be previously defined.

7 If a **specification expression** includes a specification inquiry that depends on the type, a **type parameter**, an **array bound**, or a **cobound** of an entity specified in the same *specification-part*, the type, **type parameter**, **array bound**, or **cobound** shall be specified in a prior specification of the *specification-part*. The prior specification may be to the left of the specification inquiry in the same statement, but shall not be within the same *entity-decl*. If a **specification expression** includes a reference to the value of an element of an array specified in the same *specification-part*, the array shall be completely specified in prior declarations.

8 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**.

9 A **component specification expression** is a **specification expression** in which

- there are no references to specification functions,
- there are no references to the intrinsic functions **ALLOCATED**, **ASSOCIATED**, **COMMAND_ARGUMENT_COUNT**, **EXTENDS_TYPE_OF**, **GET_TEAM**, **NUM_IMAGES**, **PRESENT**, **SAME_TYPE_AS**, **TEAM_NUMBER**, or **THIS_IMAGE**,

⁵Statement functions are obsolescent.

- every specification inquiry reference is a constant expression, and
- the value does not depend on the value of a variable.

A reference to the intrinsic function **TRANSFER** in a **component specification expression** is permitted only if each argument is a **constant expression** and each **ultimate pointer** component of the **SOURCE** argument is **disassociated**.

NOTE 2

The following are examples of **specification expressions**:

```

LBOUND (B, 1) + 5    ! B is an assumed-shape dummy array
M + LEN (C)           ! M and C are dummy arguments
2 * PRECISION (A)    ! A is a real variable made accessible by a USE statement

```

10.1.12 Constant expression

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) an enum constructor whose *expr* is a constant expression,
- (5) an enumeration constructor whose *expr* is a constant expression,
- (6) 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**,
- (7) a reference to an **elemental** standard intrinsic function, where each argument is a **constant expression**,
- (8) a reference to a standard intrinsic function that is **transformational**, other than **COMMAND_ARGUMENT_COUNT**, **GET_TEAM**, **NULL**, **NUM_IMAGES**, **TEAM_NUMBER**, **THIS_IMAGE**, or **TRANSFER**, where each argument is a **constant expression**,
- (9) 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**,
- (10) 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**,
- (11) a reference to a **transformational function** from the intrinsic module **IEEE_ARITHMETIC** or **IEEE_EXCEPTIONS**, where each argument is a **constant expression**,

- (12) a previously declared **kind type parameter** of the derived type being defined,
- (13) a **data-i-do-variable** within a **data-implied-do**,
- (14) 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
- (15) 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**.

R1030 *constant-expr* **is** *expr*

C1012 (R1030) *constant-expr* shall be a **constant expression**.

R1031 *default-char-constant-expr* **is** *default-char-expr*

C1013 (R1031) *default-char-constant-expr* shall be a **constant expression**.

R1032 *int-constant-expr* **is** *int-expr*

C1014 (R1032) *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** unless the specification inquiry appears within an **initialization**.
- 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

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 (REAL (RADIX (0.0))))

```

where A is an **explicit-shape array** with constant bounds, X is default real, and number_of_decimal_digits is an integer **named constant**.

10.2 Assignment

10.2.1 Assignment statement

10.2.1.1 General form

R1033 *assignment-stmt* **is** *variable* = *expr*

C1015 (R1033) The **variable** shall not be a whole **assumed-size array**.

NOTE

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.

10.2.1.2 Intrinsic assignment statement

- 1 An intrinsic assignment statement is an assignment statement that is not a *defined assignment* statement (10.2.1.4). In an intrinsic assignment statement,

- (1) if the variable is *polymorphic* it shall be *allocatable*, and not a *coarray* or a data object with a *coarray potential subobject component*,
- (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 not a *coarray* or of a type that has a *coarray potential subobject component*,
- (4) if the variable is *polymorphic* it shall be *type compatible* with *expr*,
- (5) if *expr* is a *boz-literal-constant*, the variable shall be of type integer or real,
- (6) if the variable is not *polymorphic* and *expr* is not a *boz-literal-constant*, the *declared types* of the variable and *expr* shall conform as specified in Table 10.8,
- (7) 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,
- (8) otherwise if the variable is of type character *expr* shall have the same *kind type parameter*,
- (9) 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
- (10) 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 10.8 — Intrinsic assignment type conformance

Type of the variable	Type of <i>expr</i>
integer	integer, real, complex
real	integer, real, complex
complex	integer, real, complex
character	character
logical	logical
derived type	same derived type as the variable
enumeration type	same enumeration type
enum type	same enum type, or integer; if of type integer, a primary in <i>expr</i> shall be an enumerator of the enum type

- 2 If the variable in an intrinsic assignment statement is a *coindexed object*,
- the variable shall not be *polymorphic*,
 - the variable shall not have an *allocatable ultimate component*,
 - the variable shall be *conformable* with *expr*, and

- each **deferred length type parameter** of the variable shall have the same value as the corresponding type parameter of *expr*.

- 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. If the variable is a **coarray** or a **coindexed object**, it shall not be an unallocated **allocatable** variable.

10.2.1.3 Interpretation of intrinsic assignments

- 1 Execution of an intrinsic assignment causes, in effect, the evaluation of the expression *expr* and all expressions within *variable* (10.1), the possible conversion of *expr* to the type and type parameters of the variable (Table 10.9), and the definition of the variable with the resulting value. The execution of the assignment shall have the same effect as if the evaluation of *expr* and the evaluation of all expressions in *variable* occurred before any portion of the variable is defined by the assignment. The evaluation of expressions within *variable* shall neither affect nor be affected by the evaluation of *expr*.
- 2 If the variable is a pointer, the value of *expr* is assigned to the **target** of the variable.
- 3 If the variable is an unallocated **allocatable** array, *expr* shall have the same **rank**. If the variable is an allocated **allocatable** variable, it is deallocated if *expr* is an array of different shape, any corresponding **length type parameter** values of the variable and *expr* differ, or the variable is **polymorphic** and the **dynamic type** or any corresponding **kind type parameter** values of the variable and *expr* differ. If the variable is or becomes an unallocated **allocatable** variable, it is then allocated with
 - the same **dynamic type** and **kind type parameter** values as *expr* if the variable is **polymorphic**,
 - each **deferred type parameter** equal to the corresponding type parameter of *expr*,
 - the same bounds as before if the variable is an array and *expr* is scalar, and
 - the shape of *expr* with each lower bound equal to the corresponding element of **LBOUND** (*expr*) if *expr* is an array.

NOTE 1

For example, given the declaration

```
CHARACTER (:), ALLOCATABLE :: NAME
```

then after the assignment statement

```
NAME = 'Dr. ' // FIRST_NAME // ' ' // SURNAME
```

NAME will have the length **LEN** (FIRST_NAME) + **LEN** (SURNAME) + 5, even if it had previously been unallocated, or allocated with a different length. However, the assignment statement

```
NAME (:) = 'Dr. ' // FIRST_NAME // ' ' // SURNAME
```

is only conforming if NAME is already allocated at the time of the assignment; the assigned value is truncated or blank padded to the previously allocated length of NAME.

- 4 Both *variable* and *expr* may contain references to any portion of the variable.

NOTE 2

For example, in the character intrinsic assignment statement:

```
STRING (2:5) = STRING (1:4)
```

the assignment of the first character of STRING to the second character does not affect the evaluation of STRING (1:4). If the value of STRING prior to the assignment was 'ABCDEF', the value following the assignment is 'AABCDF'.

- 1 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
2 the variable with every element of the array equal to the scalar value of *expr*.
- 3 6 If the variable is an array, the assignment is performed element-by-element on corresponding array ele-
4 ments of the variable and *expr*.

NOTE 3

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.

- 5 7 The processor may perform the element-by-element assignment in any order.

NOTE 4

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)
```

- 6 8 For an intrinsic assignment statement where the variable is of numeric type, the *expr* can have a different
7 *numeric type* or *kind type parameter*, in which case the value of *expr* is converted to the type and *kind type*
8 *parameter* of the variable according to the rules of Table 10.9.
- 9 9 For an intrinsic assignment statement where the variable is of type integer or real, and *expr* is a *boz-literal-*
10 *constant*, *expr* is converted to the type and *kind type parameter* of the variable according to the rules of
11 Table 10.9.

Table 10.9 — Numeric conversion and the assignment statement

Type of the variable	Value assigned
integer	INT (<i>expr</i> , KIND = KIND (<i>variable</i>))
real	REAL (<i>expr</i> , KIND = KIND (<i>variable</i>))
complex	CMPLX (<i>expr</i> , KIND = KIND (<i>variable</i>))
NOTE INT, REAL, CMPLX, and KIND are the generic names of functions defined in 16.9.	

- 12 10 For an intrinsic assignment statement where the variable is of type logical, the *expr* can have a different *kind*
13 *type parameter*, in which case the value of *expr* is converted to the *kind type parameter* of the variable.
- 14 11 For an intrinsic assignment statement where the variable is of type character, the *expr* can have a different
15 character length parameter in which case the conversion of *expr* to the length of the variable is as follows.
- 16 (1) If the length of the variable is less than that of *expr*, the value of *expr* is truncated from the right
17 until it is the same length as the variable.
- 18 (2) If the length of the variable is greater than that of *expr*, the value of *expr* is extended on the right
19 with blanks until it is the same length as the variable.
- 20 12 For an intrinsic assignment statement where the variable is of type character, if *expr* has a different *kind*

1 **type parameter**, each character *c* in *expr* is converted to the **kind type parameter** of the variable by **ACHAR** (
2 **IACHAR**(*c*), **KIND** (*variable*)).

NOTE 5

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.

3 13 For an intrinsic assignment where the variable is of enum type, if *expr* is of type integer, it is converted to
4 the type of the variable as if by the enum constructor *enum-type-name* (*expr*).

5 14 For an intrinsic assignment of the type **C_PTR** or **C_FUNPTR** from the intrinsic module **ISO_C_BINDING**, or
6 of the type **TEAM_TYPE** from the intrinsic module **ISO_FORTRAN_ENV**, the variable becomes undefined if
7 the variable and *expr* are not on the same **image**.

NOTE 6

An intrinsic assignment statement for a variable of **declared type C_PTR**, **C_FUNPTR**, or **TEAM_TYPE** cannot involve a **coindexed object**, see C915, which prevents inappropriate copying from one **image** to another. However, such copying can occur for a component in a derived-type intrinsic assignment.

8 15 An intrinsic assignment where the variable is of derived type is performed as if each component of the
9 variable were assigned from the corresponding component of *expr* using **pointer assignment** (10.2.2) for
10 each pointer component, **defined assignment** for each nonpointer nonallocatable component of a type that
11 has a **type-bound defined assignment** consistent with the component, intrinsic assignment for each other
12 nonpointer nonallocatable component, and intrinsic assignment for each allocated **coarray** component. For
13 unallocated **coarray** components, the corresponding component of the variable shall be unallocated. For a
14 noncoarray **allocatable** component the following sequence of operations is applied.

- 15 (1) If the component of the variable is allocated, it is deallocated.
- 16 (2) If the component of the value of *expr* is allocated, the corresponding component of the variable
17 is allocated with the same **dynamic type** and type parameters as the component of the value of
18 *expr*. If it is an array, it is allocated with the same bounds. The value of the component of the
19 value of *expr* is then assigned to the corresponding component of the variable using **defined**
20 **assignment** if the **declared type** of the component has a **type-bound defined assignment** con-
21 sistent with the component, and intrinsic assignment for the **dynamic type** of that component
22 otherwise.

23 16 The processor may perform the component-by-component assignment in any order or by any means that
24 has the same effect.

NOTE 7

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 assignment

$$C = D$$

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 8

If an **allocatable** component of *expr* is unallocated, the corresponding component of the variable has an allocation status of unallocated after execution of the assignment.

10.2.1.4 Defined assignment statement

1 A **defined assignment** statement is an assignment statement that is defined by a subroutine and a **generic interface** (7.5.5, 15.4.3.4.3) that specifies **ASSIGNMENT** (=).

2 A subroutine defines the **defined assignment** $x_1 = x_2$ if

- (1) the subroutine is specified with a **SUBROUTINE** (15.6.2.3) or **ENTRY**⁶ (15.6.2.6) statement that specifies two **dummy arguments**, d_1 and d_2 ,
- (2) either
 - (a) a **generic interface** (15.4.3.2) provides the subroutine with a *generic-spec* of **ASSIGNMENT** (=), or
 - (b) there is a **generic binding** (7.5.5) 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) d_1 and d_2 are type-compatible with x_1 and x_2 , respectively,
- (4) the type parameters, if any, of d_1 and d_2 match the corresponding type parameters of x_1 and x_2 , respectively, and
- (5) either
 - (a) the **ranks** of x_1 and x_2 match those of d_1 and d_2 or
 - (b) the subroutine is **elemental**, x_2 is scalar or has the same **rank** as x_1 , and there is no other subroutine that defines the assignment.

3 If d_1 or d_2 is an array, the shapes of x_1 and x_2 shall match the shapes of d_1 and d_2 , respectively. If the subroutine is **elemental**, x_2 shall be **conformable** with x_1 .

10.2.1.5 Interpretation of defined assignment statements

1 The interpretation of a **defined assignment** is provided by the subroutine that defines it.

2 If the **defined assignment** is an **elemental assignment** and the variable in the assignment is an array, the **defined assignment** is performed element-by-element, on corresponding elements of the variable and *expr*. If *expr* is a scalar, it is treated as if it were an array of the same shape as the variable with every element of the array equal to the scalar value of *expr*.

NOTE

The rules of **defined assignment** (15.4.3.4.3), procedure references (15.5), subroutine references (15.5.4), and **elemental** subroutine arguments (15.9.3) ensure that the **defined assignment** has the same effect as if the evaluation of all operations in x_2 and x_1 occurs before any portion of x_1 is defined. If an **elemental assignment** is defined by a pure **elemental** subroutine, the element assignments can be performed simultaneously or in any order.

10.2.2 Pointer assignment

10.2.2.1 General

1 **Pointer assignment** causes a pointer to become associated with a **target** or causes its **pointer association** status to become **disassociated** or undefined. Any previous association between the pointer and a **target** is broken.

⁶The **ENTRY statement** is obsolescent.

1 2 **Pointer assignment** for a pointer component of a structure can also take place by execution of a derived-
2 type **intrinsic assignment statement** (10.2.1.3).

3 10.2.2.2 Syntax of the pointer assignment statement

4 R1034 *pointer-assignment-stmt* **is** *data-pointer-object* [(*bounds-spec-list*)] => *data-target*
5 **or** *data-pointer-object* (*lower-bounds-expr* :) => *data-target*
6 **or** *data-pointer-object* (*bounds-remapping-list*) => *data-target*
7 **or** *data-pointer-object* (*lower-bounds-expr* : *upper-bounds-expr*) ■
8 ■ => *data-target*
9 **or** *proc-pointer-object* => *proc-target*

10 R1035 *data-pointer-object* **is** *variable-name*
11 **or** *scalar-variable* % *data-pointer-component-name*

12 C1016 (R1034) If *data-target* is not **unlimited polymorphic**, *data-pointer-object* shall be **type compatible**
13 (7.3.3) with it and the corresponding **kind type parameters** shall be equal.

14 C1017 (R1034) If *data-target* is **unlimited polymorphic**, *data-pointer-object* shall be **unlimited**
15 **polymorphic**, or of a type with the **BIND** attribute or the **SEQUENCE** attribute.

16 C1018 (R1034) If *bounds-spec-list* is specified, the number of *bounds-specs* shall equal the **rank** of *data-*
17 *pointer-object*.

18 C1019 (R1034) If *bounds-remapping-list* is specified, the number of *bounds-remappings* shall equal the
19 **rank** of *data-pointer-object*.

20 C1020 If *lower-bounds-expr* and *upper-bounds-expr* appear in a *pointer-assignment-stmt*, at least one of
21 them shall be a rank-one array of constant size equal to the rank of *data-pointer-object*.

22 C1021 If *lower-bounds-expr* appears in a *pointer-assignment-stmt* but not *upper-bounds-expr*, it shall be a
23 rank-one array of constant size equal to the rank of *data-pointer-object*.

24 C1022 If neither *bounds-remapping-list* nor *upper-bounds-expr* appears in a *pointer-assignment-stmt*, the
25 **ranks** of *data-pointer-object* and *data-target* shall be the same.

26 C1023 (R1034) A coarray *data-target* shall have the **VOLATILE** attribute if and only if the *data-pointer-*
27 *object* has the **VOLATILE** attribute.

28 C1024 (R1035) A *variable-name* shall have the **POINTER** attribute.

29 C1025 (R1035) A *scalar-variable* shall be a *data-ref*.

30 C1026 (R1035) A *data-pointer-component-name* shall be the name of a component of *scalar-variable* that
31 is a data pointer.

32 C1027 (R1035) A *data-pointer-object* shall not be a **coindexed object**.

33 R1036 *bounds-spec* **is** *lower-bound-expr* :

34 R1037 *bounds-remapping* **is** *lower-bound-expr* : *upper-bound-expr*

35 R1038 *data-target* **is** *expr*

36 C1028 (R1038) 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**.

C1029 (R1038) A **data-target** shall not be a **coindexed object**.

NOTE

A **data pointer** and its **target** are always on the same **image**. A **coarray** can be of a derived type with pointer or **allocatable subcomponents**. For example, if PTR is a pointer component, and Z%PTR on **image** P has been associated with a **target** by execution of an **ALLOCATE statement** or a **pointer assignment** on **image** P, Z[P]%PTR will be a reference to that **target**.

R1039 *proc-pointer-object* **is** *proc-pointer-name*
 or *proc-component-ref*

R1040 *proc-component-ref* **is** *scalar-variable* % *procedure-component-name*

C1030 (R1040) The *scalar-variable* shall be a **data-ref** that is not a **coindexed object**.

C1031 (R1040) The *procedure-component-name* shall be the name of a **procedure pointer** component of the **declared type** of *scalar-variable*.

R1041 *proc-target* **is** *expr*
 or *procedure-name*
 or *proc-component-ref*

C1032 (R1041) An *expr* shall be a reference to a function whose result is a **procedure pointer**.

C1033 (R1041) A *procedure-name* shall be the name of an **internal**, **module**, or **dummy** procedure, a **procedure pointer**, a **specific** intrinsic function⁷ listed in Table 16.2, or an **external procedure** that is accessed by **use** or **host** association, referenced in the **scoping unit** as a procedure, or that has the **EXTERNAL** attribute.

C1034 (R1041) The *proc-target* shall not be a nonintrinsic **elemental procedure**.

1 In a pointer assignment statement, *data-pointer-object* or *proc-pointer-object* denotes the pointer object and *data-target* or *proc-target* denotes the pointer target.

2 For pointer assignment performed by a derived-type **intrinsic assignment statement**, the pointer object is the pointer component of the variable and the pointer target is the corresponding component of *expr*.

10.2.2.3 Data pointer assignment

1 If the pointer object is not polymorphic (7.3.2.3) and the pointer target is **polymorphic** with **dynamic type** that differs from its **declared type**, the assignment target is the ancestor component of the pointer target that has the type of the pointer object. Otherwise, the assignment target is the pointer target.

2 If the pointer target is not a pointer, the pointer object becomes **pointer associated** with the assignment target; if the pointer target is a pointer with a target that is not on the same **image**, the **pointer association** status of the pointer object becomes undefined. Otherwise, the **pointer association** status of the pointer object becomes that of the pointer target; if the pointer target is associated with an object, the pointer object becomes associated with the assignment target. If the pointer target is **allocatable**, it shall be allocated.

⁷Specific intrinsic function names are obsolescent.

NOTE

A [pointer assignment statement](#) is not permitted to involve a [coindexed](#) pointer or target, see C1027 and C1029. 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.

- 1 3 If the pointer object is [polymorphic](#), it assumes the [dynamic type](#) of the pointer target. If the pointer object
2 is of a type with the [BIND attribute](#) or the [SEQUENCE attribute](#), the [dynamic type](#) of the pointer target shall
3 be that type.
- 4 4 If the pointer target is a [disassociated](#) pointer, all nondeferred type parameters of the [declared type](#) of
5 the pointer object that correspond to nondeferred type parameters of the pointer target shall have the
6 same values as the corresponding type parameters of the pointer target. Otherwise, all nondeferred type
7 parameters of the [declared type](#) of the pointer object shall have the same values as the corresponding type
8 parameters of the pointer target.
- 9 5 If the pointer object has nondeferred type parameters that correspond to [deferred type parameters](#) of the
10 pointer target, the pointer target shall not be a pointer with undefined association status.
- 11 6 If the pointer object has the [CONTIGUOUS attribute](#), the pointer target shall be [contiguous](#).
- 12 7 If the target of a pointer is a [coarray](#), the pointer shall have the [VOLATILE attribute](#) if and only if the [coarray](#)
13 has the [VOLATILE attribute](#).
- 14 8 If [bounds-remapping-list](#) appears, it specifies the upper and lower [bounds](#) of each dimension of the pointer,
15 and thus the [extents](#); the pointer target shall be [simply contiguous](#) (9.5.4) or of [rank](#) one, and shall not be
16 a [disassociated](#) or [undefined](#) pointer. The number of elements of the pointer target shall not be less than
17 the number implied by the [bounds-remapping-list](#). The elements of the pointer object are associated with
18 those of the pointer [target](#), in array element order; if the pointer target has more elements than specified
19 for the pointer object, the remaining elements are not associated with the pointer object.
- 20 9 If [lower-bounds-expr](#) and [upper-bounds-expr](#) appear, the effect is the same as a [bounds-remapping-list](#) with
21 each [bounds-remapping](#) comprising corresponding elements of the lower and upper bounds arrays, in array
22 element order. If one of them is a scalar, the effect is as if it were broadcast to the same shape as the other.
- 23 10 If neither [bounds-remapping-list](#) nor [upper-bounds-expr](#) appears, the extent of a dimension of the pointer
24 object is the extent of the corresponding dimension of the pointer target. If [bounds-spec-list](#) or [lower-](#)
25 [bounds-expr](#) appears, it specifies the lower bounds; otherwise, the lower bound of each dimension is the
26 result of the intrinsic function [LBOUND](#) (16.9.119) applied to the corresponding dimension of the pointer
27 target. The upper bound of each dimension is one less than the sum of the lower bound and the extent.

10.2.2.4 Procedure pointer assignment

- 29 1 If the pointer target is not a pointer or [dummy argument](#), the pointer object becomes [pointer associated](#)
30 with the pointer target. If the pointer target is a nonpointer [dummy argument](#), the pointer object becomes
31 associated with the [ultimate argument](#) of the [dummy argument](#). Otherwise, the [pointer association](#) status
32 of the pointer object becomes that of the pointer target; if the pointer target is associated with a procedure,
33 the pointer object becomes associated with the same procedure.
- 34 2 The [host instance](#) (15.6.2.4) of an associated [procedure pointer](#) is the [host instance](#) of its target.
- 35 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**, the pointer target may be **simple** even if the pointer object is not **simple**, and the pointer target may be an **elemental** intrinsic procedure, even though the pointer object cannot be **elemental**.

4 If the **characteristics** of the pointer object or the pointer target are such that an **explicit interface** is required, both the pointer object and the pointer target shall have an **explicit interface**.

5 If the pointer object has an **implicit interface** and is explicitly typed or referenced as a function, the pointer target shall be a function. If the pointer object has an **implicit interface** and is referenced as a subroutine, the pointer target shall be a subroutine.

6 If the pointer object is a function with an **implicit interface**, the pointer target shall be a function with the same type; corresponding type parameters shall have the same value.

7 If *procedure-name* is a specific procedure name that is also a generic name, only the specific procedure is associated with the pointer object.

10.2.2.5 Examples of pointer assignment statements

NOTE 1

The following are examples of pointer assignment statements. (See 15.4.3.6, NOTE 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, BESSEL is a procedure with a compatible interface.
P => BESSEL

! Likewise for a structure component.
STRUCT % COMPONENT => BESSEL
```

NOTE 2

It is possible to obtain different **rank** views of parts of an object by specifying upper bounds in pointer assignment statements. This requires that the object be either **rank** one or **contiguous**. Consider the following example, in which a matrix is under consideration. The matrix is stored as a rank-one object in MYDATA because its diagonal is needed for some reason – the diagonal cannot be gotten as a single object from a rank-two representation. The matrix is represented as a rank-two view of MYDATA.

```
real, target :: MYDATA ( NR*NC )      ! An automatic array
real, pointer :: MATRIX ( :, : )      ! A rank-two view of MYDATA
real, pointer :: VIEW_DIAG ( : )
MATRIX (1:NR, 1:NC) => MYDATA          ! The MATRIX view of the data
VIEW_DIAG => MYDATA (1::NR+1)          ! The diagonal of MATRIX
```

Rows, columns, or blocks of the matrix can be accessed as sections of MATRIX.

Rank remapping can be applied to CONTIGUOUS arrays, for example:

```
REAL, CONTIGUOUS, POINTER :: A (:)
REAL, CONTIGUOUS, TARGET  :: B (:,:) ! Dummy argument
A (1:SIZE(B)) => B                  ! Linear view of a rank-2 array
```

10.2.3 Masked array assignment – WHERE

10.2.3.1 General form of the masked array assignment

- 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.

R1042 *where-stmt* **is** WHERE (*mask-expr*) *where-assignment-stmt*

R1043 *where-construct* **is** *where-construct-stmt*
 [*where-body-construct*] ...
 [*masked-elsewhere-stmt*
 [*where-body-construct*] ...] ...
 [*elsewhere-stmt*
 [*where-body-construct*] ...]
 end-where-stmt

R1044 *where-construct-stmt* **is** [*where-construct-name*:] WHERE (*mask-expr*)

R1045 *where-body-construct* **is** *where-assignment-stmt*
 or *where-stmt*
 or *where-construct*

R1046 *where-assignment-stmt* **is** *assignment-stmt*

R1047 *mask-expr* **is** *logical-expr*

R1048 *masked-elsewhere-stmt* **is** ELSEWHERE (*mask-expr*) [*where-construct-name*]

R1049 *elsewhere-stmt* **is** ELSEWHERE [*where-construct-name*]

R1050 *end-where-stmt* **is** END WHERE [*where-construct-name*]

C1035 (R1046) A *where-assignment-stmt* that is a **defined assignment** shall be **elemental**.

C1036 (R1043) 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*.

C1037 (R1045) 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

Examples of 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.
```

NOTE (cont.)

END WHERE

10.2.3.2 Interpretation of masked array assignments

1 When a WHERE statement or a *where-construct-stmt* is executed, a control mask is established. In addition, when a WHERE construct statement is executed, a pending control mask is established. If the statement does not appear as part of a *where-body-construct*, the *mask-expr* of the statement is evaluated, and the control mask is established to be the value of *mask-expr*. The pending control mask is established to have the value *.NOT. mask-expr* upon execution of a WHERE construct statement that does not appear as part of a *where-body-construct*.

2 The *mask-expr* in a WHERE statement, WHERE construct statement, or masked ELSEWHERE statement, is evaluated at most once per execution of the statement.

3 Each statement in a WHERE construct is executed in sequence.

4 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*.

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 1

The establishment of control masks and the pending control mask is illustrated with the following example:

```
WHERE (cond1)           ! Statement 1
...
ELSEWHERE (cond2)       ! Statement 2
...
ELSEWHERE               ! Statement 3
...
END WHERE
```

Following execution of statement 1, the control mask has the value cond1 and the pending control mask has the value *.NOT. cond1*. Following execution of statement 2, the control mask has the value *(.NOT. cond1) .AND. cond2* and the pending control mask has the value *(.NOT. cond1) .AND. (.NOT. cond2)*. Following execution of statement 3, the control mask has the value *(.NOT. cond1) .AND. (.NOT. cond2)*. The false condition values are propagated through the execution of the masked ELSEWHERE statement.

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-*

1 *expr*, the function is evaluated without any masked control; that is, all of its argument expressions are fully
2 evaluated and the function is fully evaluated. If the result is an array and the reference is not within the
3 argument list of a non**elemental** function, elements corresponding to true values in the control mask are
4 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*
6 or in a *mask-expr*, and is not within the argument list of a non**elemental** function reference, the operation
7 is performed or the function is evaluated only for the elements corresponding to true values of the control
8 mask.

9 11 If an array constructor appears in a *where-assignment-stmt* or in a *mask-expr*, the array constructor is eval-
10 uated without any masked control and then the *where-assignment-stmt* is executed or the *mask-expr* is
11 evaluated.

12 When a *where-assignment-stmt* is executed, the values of *expr* that correspond to true values of the control
13 mask are assigned to the corresponding elements of the variable.

14 13 The value of the control mask is established by the execution of a WHERE statement, a WHERE construct
15 statement, an ELSEWHERE statement, a masked ELSEWHERE statement, or an ENDWHERE statement.
16 Subsequent changes to the value of entities in a *mask-expr* have no effect on the value of the control mask.
17 The execution of a function reference in the mask expression of a WHERE statement is permitted to affect
18 entities in the assignment statement.

NOTE 2

Examples of function references in masked array assignments are:

[illegible]19 **10.2.4 FORALL**

10.2.4.1 Obsolescence

21 1 The FORALL construct and statement are obsolescent.

22 10.2.4.2 Form of the FORALL Construct

1 The FORALL construct allows multiple assignments, masked array (WHERE) assignments, and nested FOR-
ALL constructs and statements to be controlled by a single *concurrent-control-list* and *scalar-mask-expr*.

```

25   R1051 forall-construct      is forall-construct-stmt
26                               [forall-body-construct ] ...
27                               end-forall-stmt

```

```

28      R1052 forall-construct-stmt      is [forall-construct-name :] FORALL concurrent-header

```

```

29      R1053 forall-body-construct      is forall-assignment-stmt
30                                     or where-stmt
31                                     or where-construct
32                                     or forall-construct
33                                     or forall-stmt

```

1 R1054 *forall-assignment-stmt* is *assignment-stmt*
 2 or *pointer-assignment-stmt*

3 R1055 *end-forall-stmt* is END FORALL [*forall-construct-name*]

4 C1038 (R1055) If the *forall-construct-stmt* has a *forall-construct-name*, the *end-forall-stmt* shall have the
 5 same *forall-construct-name*. If the *end-forall-stmt* has a *forall-construct-name*, the *forall-construct-*
 6 *stmt* shall have the same *forall-construct-name*.

7 C1039 A statement in a *forall-body-construct* shall not define an *index-name* of the *forall-construct*.

8 C1040 (R1053) Any procedure referenced in a *forall-body-construct*, including one referenced by a defined
 9 operation, assignment, or *finalization*, shall be a *pure procedure*.

10 C1041 (R1053) A *forall-body-construct* shall not be a branch target.

11 2 The scope and attributes of an *index-name* in a *concurrent-header* in a FORALL construct or statement are
 12 described in 19.4.

13 10.2.4.3 Execution of the FORALL construct

14 10.2.4.3.1 Execution stages

15 1 There are three stages in the execution of a FORALL construct:

- 16 (1) determination of the values for *index-name* variables,
- 17 (2) evaluation of the *scalar-mask-expr*, and
- 18 (3) execution of the FORALL body constructs.

19 10.2.4.3.2 Determination of the values for index variables

20 1 The values of the index variables are determined as they are for the *DO CONCURRENT statement*
 21 (11.1.7.4.2).

22 10.2.4.3.3 Evaluation of the mask expression

23 1 The mask expression is evaluated as it is for the *DO CONCURRENT statement* (11.1.7.4.2).

24 10.2.4.3.4 Execution of the FORALL body constructs

25 1 The *forall-body-constructs* are executed in the order in which they appear. Each construct is executed for
 26 all active combinations of the *index-name* values with the following interpretation:

27 2 Execution of a *forall-assignment-stmt* that is an *assignment-stmt* causes the evaluation of *expr* and all ex-
 28 pressions within *variable* for all active combinations of *index-name* values. These evaluations may be done
 29 in any order. After all these evaluations have been performed, each *expr* value is assigned to the corres-
 30 ponding *variable*. The assignments may occur in any order.

31 3 Execution of a *forall-assignment-stmt* that is a *pointer-assignment-stmt* causes the evaluation of all expres-
 32 sions within *data-target* and *data-pointer-object* or *proc-target* and *proc-pointer-object*, the determination
 33 of any pointers within *data-pointer-object* or *proc-pointer-object*, and the determination of the *target* for
 34 all active combinations of *index-name* values. These evaluations may be done in any order. After all these
 35 evaluations have been performed, each *data-pointer-object* or *proc-pointer-object* is associated with the
 36 corresponding *target*. These associations may occur in any order.

- 1 4 In a *forall-assignment-stmt*, a *defined assignment* subroutine shall not reference any *variable* that becomes
2 defined by the statement.

NOTE

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.

- 3 5 Each statement in a *where-construct* (10.2.3) within a *forall-construct* is executed in sequence. When a
4 *where-stmt*, *where-construct-stmt* or *masked-elsewhere-stmt* is executed, the statement's *mask-expr* is eval-
5 uated for all active combinations of *index-name* values as determined by the outer *forall-constructs*, masked
6 by any control mask corresponding to outer *where-constructs*. A *where-assignment-stmt* is executed for all
7 active combinations of *index-name* values, masked by the control mask in effect for the *where-assignment-*
8 *stmt*.
- 9 6 Execution of a *forall-stmt* or *forall-construct* causes the evaluation of the *concurrent-limit* and *concurrent-*
10 *step* expressions in the *concurrent-control-list* for all active combinations of the *index-name* values of the
11 outer FORALL construct. The set of combinations of *index-name* values for the inner FORALL is the union of
12 the sets defined by these limits and steps for each active combination of the outer *index-name* values; it also
13 includes the outer *index-name* values. The *scalar-mask-expr* is then evaluated for all combinations of the
14 *index-name* values of the inner construct to produce a set of active combinations for the inner construct. If
15 there is no *scalar-mask-expr*, it is as if it appeared with the value true. Each statement in the inner FORALL
16 is then executed for each active combination of the *index-name* values.

17 10.2.4.4 The FORALL statement

- 18 1 The FORALL statement allows a single assignment statement or *pointer assignment statement* to be con-
19 trolled by a set of index values and an optional mask expression.

20 R1056 *forall-stmt* is FORALL *concurrent-header forall-assignment-stmt*

- 21 2 A FORALL statement is equivalent to a FORALL construct containing a single *forall-body-construct* that is a
22 *forall-assignment-stmt*.

- 23 3 The scope of an *index-name* in a *forall-stmt* is the statement itself (19.4).

24 10.2.4.5 Restrictions on FORALL constructs and statements

- 25 1 A many-to-one assignment is more than one assignment to the same object, or association of more than
26 one *target* with the same pointer, whether the object is referenced directly or indirectly through a pointer.
27 A many-to-one assignment shall not occur within a single statement in a FORALL construct or statement.
28 It is possible to assign or pointer-assign to the same object in different *assignment* or *pointer assignment*
29 statements in a FORALL construct.

NOTE

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.

- 1 2 Within the scope of a FORALL construct, a nested FORALL statement or FORALL construct shall not have the
2 same *index-name*. The *concurrent-header* expressions within a nested FORALL may depend on the values
3 of outer *index-name* variables.

11 Execution control

11.1 Executable constructs containing blocks

11.1.1 Blocks

1 The following are executable constructs that contain blocks:

- ASSOCIATE construct;
- BLOCK construct;
- CHANGE TEAM construct;
- CRITICAL construct;
- DO construct;
- IF construct;
- SELECT CASE construct;
- SELECT RANK construct;
- SELECT TYPE construct.

R1101 *block* is [*execution-part-construct*] ...

2 Executable constructs can 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

An example of a construct containing a block is:

```
IF (A > 0.0) THEN
  B = SQRT (A)    ! These two statements
  C = LOG (A)     ! form a block.
END IF
```

11.1.2 Rules governing blocks

11.1.2.1 Control flow in blocks

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 (15.5.3, 15.5.4) may appear in a block.

11.1.2.2 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

- execution of the last executable construct in the block completes without branching to a statement within the block,

- a branch (11.2) within the block that has a branch target outside the block occurs,
- a RETURN statement within the block is executed, or
- an EXIT or CYCLE statement that belongs to a construct that contains the block is executed.

NOTE

The action that takes place at the terminal boundary depends on the particular construct and on the block within that construct.

11.1.3 ASSOCIATE construct**11.1.3.1 Purpose and form of the ASSOCIATE construct**

- 1 The ASSOCIATE construct associates named entities with expressions or variables during the execution of its block. These named construct entities (19.4) are associating entities (19.5.1.6). The names are associate names.

R1102 *associate-construct* **is** *associate-stmt*
 block
 end-associate-stmt

R1103 *associate-stmt* **is** [*associate-construct-name* :] ASSOCIATE ■
 ■ (*association-list*)

R1104 *association* **is** *associate-name* => *selector*

R1105 *selector* **is** *expr*
 or *variable*

C1101 (R1104) If *selector* is not a *variable* or is a *variable* that has a *vector subscript*, neither *associate-name* nor any subobject thereof shall appear in a variable definition context (19.6.7) or pointer association context (19.6.8).

C1102 (R1104) An *associate-name* shall not be the same as another *associate-name* in the same *associate-stmt*.

C1103 (R1105) *variable* shall not be a *coindexed object*.

C1104 (R1105) *expr* shall not be a variable.

C1105 (R1105) *expr* shall not be a *designator* of a *procedure pointer* or a function reference that returns a *procedure pointer*.

R1106 *end-associate-stmt* **is** END ASSOCIATE [*associate-construct-name*]

C1106 (R1106) 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*.

11.1.3.2 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 (19.5.1.6) with the corresponding selector. The **associating entity** assumes the **declared type** and type parameters of the selector. If and only if the selector is **polymorphic**, the **associating entity** is **polymorphic**.

The other attributes of the **associating entity** are described in 11.1.3.3.

It is permissible to branch to an *end-associate-stmt* only from within its ASSOCIATE construct.

11.1.3.3 Other attributes of associate names

Within an ASSOCIATE, **CHANGE TEAM**, or **SELECT TYPE** construct, each **associating entity** has the same **rank** as its associated selector. The lower bound of each dimension is the result of the intrinsic function **LBOUND** (16.9.119) applied to the corresponding dimension of **selector**. The upper bound of each dimension is one less than the sum of the lower bound and the extent. The **associating entity** does not have the **ALLOCATABLE** or **POINTER** attributes; it has the **TARGET attribute** if and only if the selector is a variable and has either the **TARGET** or **POINTER** attribute.

Within an ASSOCIATE, **SELECT RANK**, or **SELECT TYPE** construct, each **associating entity** has the same **corank** as its associated selector. If the selector is a **coarray**, the **cobounds** of each **codimension** of the **associating entity** are the same as those of the selector.

Within a **CHANGE TEAM construct**, the associating entity is a **coarray**. Its **corank** and **cobounds** are as specified in its *codimension-decl*.

Within an ASSOCIATE, **CHANGE TEAM**, **SELECT RANK**, or **SELECT TYPE** construct, the **associating entity** has the **ASYNCHRONOUS** or **VOLATILE** attribute if and only if the selector is a variable and has the attribute. If the **associating entity** is **polymorphic**, it assumes the **dynamic type** and **type parameter** values of the selector. The associating entity does not have the **OPTIONAL attribute**. If the selector has the **OPTIONAL attribute**, it cannot be absent (15.5.2.13). The **associating entity** is **contiguous** if and only if the selector is **contiguous**.

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 a selector is not permitted to appear in a variable definition context (19.6.7), neither the **associate name** nor any subobject thereof shall appear in a variable definition context or pointer association context (19.6.8).

11.1.3.4 Examples of the ASSOCIATE construct

NOTE

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.

NOTE (cont.)

```

ASSOCIATE ( W => RESULT(I,J)%W, ZX => AX%B(I,J)%D, ZY => AY%B(I,J)%D )
  W = ZX*X + ZY*Y
END ASSOCIATE

```

11.1.4 BLOCK construct

1 The BLOCK construct is an executable construct that can contain declarations.

R1107 *block-construct* **is** *block-stmt*
 [*block-specification-part*]
 block
 end-block-stmt

R1108 *block-stmt* **is** [*block-construct-name* :] BLOCK

R1109 *block-specification-part* **is** [*use-stmt*] ...
 [*import-stmt*] ...
 [*declaration-construct*] ...

R1110 *end-block-stmt* **is** END BLOCK [*block-construct-name*]

C1107 (R1107) A *block-specification-part* shall not contain a **COMMON**, **EQUIVALENCE**, **INTENT**, **NAMelist**, **OPTIONAL**, **statement function**, or **VALUE** statement.¹

C1108 (R1107) A **SAVE statement** in a BLOCK construct shall contain a *saved-entity-list* that does not specify a *common-block-name*.

C1109 The *block* of a *block-construct* shall not begin with a **FORMAT statement** or a **DATA statement**.

C1110 (R1107) If the *block-stmt* of a *block-construct* specifies a *block-construct-name*, the corresponding *end-block-stmt* shall specify the same *block-construct-name*. If the *block-stmt* does not specify a *block-construct-name*, the corresponding *end-block-stmt* shall not specify a *block-construct-name*.

2 Except for the **ASYNCHRONOUS**, **IMPORT**, and **VOLATILE** statements, specifications in a BLOCK construct declare **construct entities** whose scope is that of the BLOCK construct (19.4). The appearance of the name of an object that is not a **construct entity** in an **ASYNCHRONOUS** or **VOLATILE** statement in a BLOCK construct specifies that the object has the attribute within the construct even if it does not have the attribute outside the construct.

3 Execution of a BLOCK construct causes evaluation of the **specification expressions** within its specification part in a processor-dependent order, followed by execution of its block.

4 It is permissible to branch to an *end-block-stmt* only from within its BLOCK construct.

NOTE

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

```

¹Common blocks, **EQUIVALENCE statements**, and statement functions are obsolescent.

NOTE (cont.)

```
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
```

11.1.5 CHANGE TEAM construct**11.1.5.1 Purpose and form of the CHANGE TEAM construct**

- 1 The CHANGE TEAM construct changes the **current team**. Named **construct entities** (19.4) can be associated (19.5.1.6) with **coarrays** in the containing **scoping unit**, in the same way as for the **ASSOCIATE construct**.

R1111 *change-team-construct* **is** *change-team-stmt*
 block
 end-change-team-stmt

R1112 *change-team-stmt* **is** [*team-construct-name* :] CHANGE TEAM (*team-value* ■
 ■ [, *coarray-association-list*] [, *sync-stat-list*])

R1113 *coarray-association* **is** *codimension-decl* => *selector*

R1114 *end-change-team-stmt* **is** END TEAM [([*sync-stat-list*])] [*team-construct-name*]

R1115 *team-value* **is** *scalar-expr*

C1111 A branch (11.2) within a CHANGE TEAM construct shall not have a branch target that is outside the construct.

C1112 A **RETURN statement** shall not appear within a CHANGE TEAM construct.

C1113 If the *change-team-stmt* of a *change-team-construct* specifies a *team-construct-name*, the corresponding *end-change-team-stmt* shall specify the same *team-construct-name*. If the *change-team-stmt* of a *change-team-construct* does not specify a *team-construct-name*, the corresponding *end-change-team-stmt* shall not specify a *team-construct-name*.

C1114 In a *change-team-stmt*, a *coarray-name* in a *codimension-decl* shall not be the same as a *selector*, or another *coarray-name*, in that statement.

1 C1115 A *team-value* shall be of type *TEAM_TYPE* from the intrinsic module *ISO_FORTRAN_ENV*.

2 C1116 No *selector* shall appear more than once in a given *change-team-stmt*.

3 C1117 A *selector* in a *coarray-association* shall be a named coarray.

4 2 Each *coarray-name* in a *codimension-decl* in the CHANGE TEAM statement is an *associate name* which is
5 associated with the corresponding selector. Each *associating entity* assumes the type and *type paramet-*
6 *ers* of its selector; it is polymorphic if and only if the selector is polymorphic. The other attributes of the
7 *associating entities* are described in 11.1.3.3.

8 11.1.5.2 Execution of a CHANGE TEAM construct

9 1 The *team-values* on the *active images* that execute the CHANGE TEAM statement shall be those of *team vari-*
10 *ables* defined by corresponding executions of the same *FORM TEAM statement* (11.7.9). When the CHANGE
11 TEAM statement is executed, the *current team* shall be the team that was current when those *team vari-*
12 *ables* were defined. The current team for the statements of the CHANGE TEAM *block* is the team identified
13 by the *team-value*. If *team-value* is a variable, the variable shall not be defined or become undefined during
14 execution of the CHANGE TEAM construct. A CHANGE TEAM construct completes execution by executing
15 its END TEAM statement, which restores the *current team* to the *original team that was current* for the
16 CHANGE TEAM statement.

17 2 Execution of a CHANGE TEAM construct causes evaluation of the expressions within each *codimension-decl*
18 in the CHANGE TEAM statement, followed by execution of its block. Each *selector* shall be an *established*
19 *coarray* when the CHANGE TEAM statement begins execution.

20 3 It is permissible to branch to an *end-change-team-stmt* only from within its CHANGE TEAM construct.

21 4 An allocatable *coarray* that was allocated immediately before executing a CHANGE TEAM statement shall
22 not be deallocated during execution of the construct. An allocatable *coarray* that was unallocated imme-
23 diately before executing a CHANGE TEAM statement, and which is allocated immediately before executing
24 the corresponding END TEAM statement, is deallocated by the execution of the END TEAM statement.

25 5 Successful execution of a CHANGE TEAM statement performs an implicit synchronization of all *images* of
26 the new *team* that is identified by *team-value*. All *active images* of the new *team* shall execute the same
27 CHANGE TEAM statement. On each *image* of the new *team*, execution of the *segment* following the CHANGE
28 TEAM statement is delayed until all other *images* of that *team* have executed the same statement the same
29 number of times in the original *team*.

30 6 If the new *team* contains a *failed image* and no other error condition occurs, there is an implicit synchroniz-
31 ation of all *active images* of the new *team*. On each *active image* of the new *team*, execution of the *segment*
32 following the CHANGE TEAM statement is delayed until all other *active images* of that *team* have executed
33 the same statement the same number of times in the original *team*.

34 7 If no error condition other than the new *team* containing a *failed image* occurs, the *segments* that executed
35 before the CHANGE TEAM statement on an *active image* of the new *team* precede the *segments* that execute
36 after the CHANGE TEAM statement on another *active image* of that team.

37 8 When a CHANGE TEAM construct completes execution, there is an implicit synchronization of all *active*
38 *images* in the new *team*. On each *active image* of the new *team*, execution of the *segment* following the
39 END TEAM statement is delayed until all other *active images* of this *team* have executed the same construct
40 the same number of times in this team. The *segments* that executed before the END TEAM statement on an
41 *active image* of the new *team* precede the *segments* that execute after the END TEAM statement on another

1 [active image](#) of that [team](#).

NOTE 1

Deallocation of an allocatable coarray that was not allocated at the beginning of a CHANGE TEAM construct, but is allocated at the end of execution of the construct, occurs even for allocatable coarrays with the SAVE attribute.

NOTE 2

Execution of a CHANGE TEAM statement includes a synchronization of the executing [image](#) with the other [images](#) that will be in the same [team](#) after execution of the CHANGE TEAM statement. Synchronization of these [images](#) occurs again when the corresponding END TEAM statement is executed.

If it is desired to synchronize all of the [images](#) in the [team that was current](#) when the CHANGE TEAM statement was executed, a [SYNC TEAM statement](#) that specifies the [parent team](#) can be executed immediately after the CHANGE TEAM statement. If similar semantics are desired following the END TEAM statement, a [SYNC ALL statement](#) could immediately follow the END TEAM statement.

NOTE 3

A [coarray](#) that is established when a CHANGE TEAM statement is executed retains its [corank](#) and [cobounds](#) inside the block. If it is desired to perform remote accesses based on [corank](#) or [cobounds](#) different from those of the original [coarray](#), an [associating coarray](#) can be used. An example of this is in [C.7.7](#).

2 11.1.6 CRITICAL construct

3 1 A CRITICAL construct limits execution of a block to one [image](#) at a time.

4 R1116 *critical-construct* is [critical-stmt](#)
5 [block](#)
6 [end-critical-stmt](#)

7 R1117 *critical-stmt* is [*critical-construct-name* :] CRITICAL [([[sync-stat-list](#)])]

8 R1118 *end-critical-stmt* is END CRITICAL [*critical-construct-name*]

9 C1118 (R1116) If the [critical-stmt](#) of a [critical-construct](#) specifies a *critical-construct-name*, the corres-
10 ponding [end-critical-stmt](#) shall specify the same *critical-construct-name*. If the [critical-stmt](#) of a
11 [critical-construct](#) does not specify a *critical-construct-name*, the corresponding [end-critical-stmt](#)
12 shall not specify a *critical-construct-name*.

13 C1119 (R1116) The [block](#) of a [critical-construct](#) shall not contain a [RETURN statement](#) or an [image control](#)
14 [statement](#).

15 C1120 A branch (11.2) within a [CRITICAL construct](#) shall not have a branch target that is outside the
16 construct.

17 2 Execution of the CRITICAL construct is completed when execution of its block is completed, or the executing
18 image fails (5.3.6). A procedure invoked, directly or indirectly, from a CRITICAL construct shall not execute
19 an [image control statement](#).

20 3 The processor shall ensure that once an [image](#) has commenced executing [block](#), no other [image](#) shall com-
21 mence executing [block](#) until this [image](#) has completed execution of the construct. The [image](#) shall not ex-
22 ecute an [image control statement](#) during the execution of [block](#). The sequence of executed statements is
23 therefore a [segment](#) (11.7.2). If [image](#) M completes execution of the construct without failing and [image](#) T
24 is the next to execute the construct, the [segment](#) on [image](#) M precedes the [segment](#) on [image](#) T. Otherwise,

- 1 if **image** M completes execution of the construct by failing, and **image** T is the next to execute the construct,
- 2 the previous **segment** on **image** M precedes the **segment** on **image** T.
- 3 4 The effect of a **STAT=** or **ERRMSG=** specifier in a CRITICAL statement is specified in 11.7.11.
- 4 5 It is permissible to branch to an *end-critical-stmt* only from within its CRITICAL construct.

NOTE 1

If more than one **image** executes the block of a CRITICAL construct without failing, its execution by one **image** always either precedes or succeeds its execution by another nonfailed **image**. Typically no other statement ordering is needed. Consider the following example:

```
CRITICAL
  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 2

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
```

5 11.1.7 DO construct

6 11.1.7.1 Purpose and form of the DO construct

- 7 1 The DO construct specifies the repeated execution of a sequence of executable constructs. Such a repeated
- 8 sequence is called a loop.
- 9 2 The number of iterations of a loop can be determined at the beginning of execution of the DO construct,
- 10 or can be left indefinite ("DO forever" or DO WHILE). The execution order of the iterations can be left in-
- 11 determinate (DO CONCURRENT); except in this case, the loop can be terminated immediately (11.1.7.4.5).
- 12 An iteration of the loop can be curtailed by executing a **CYCLE statement** (11.1.7.4.4).
- 13 3 There are three phases in the execution of a DO construct: initiation of the loop, execution of each iteration
- 14 of the loop, and termination of the loop.
- 15 4 The scope and attributes of an *index-name* in a *concurrent-header* (DO CONCURRENT) are described in
- 16 19.4.

17 11.1.7.2 Form of the DO construct

18 R1119 *do-construct* is *do-stmt*

1		<i>block</i>	
2		<i>end-do</i>	
3	R1120 <i>do-stmt</i>	is <i>nonlabel-do-stmt</i>	
4		or <i>label-do-stmt</i>	
5	R1121 <i>label-do-stmt</i>	is [<i>do-construct-name</i> :] DO <i>label</i> [<i>loop-control</i>]	
6	R1122 <i>nonlabel-do-stmt</i>	is [<i>do-construct-name</i> :] DO [<i>loop-control</i>]	
7	R1123 <i>loop-control</i>	is [,] <i>do-variable</i> = <i>scalar-int-expr</i> , <i>scalar-int-expr</i> ■	
8		■ [, <i>scalar-int-expr</i>]	
9		or [,] WHILE (<i>scalar-logical-expr</i>)	
10		or [,] CONCURRENT <i>concurrent-header</i> <i>concurrent-locality</i>	
11	R1124 <i>do-variable</i>	is <i>scalar-int-variable-name</i>	
12	C1121 (R1124) The <i>do-variable</i> shall be a variable of type integer.		
13	R1125 <i>concurrent-header</i>	is ([<i>integer-type-spec</i> ::] <i>concurrent-control-list</i> ■	
14		■ [, <i>scalar-mask-expr</i>])	
15	R1126 <i>concurrent-control</i>	is <i>index-name</i> = <i>concurrent-limit</i> : <i>concurrent-limit</i> ■	
16		■ [: <i>concurrent-step</i>]	
17	R1127 <i>concurrent-limit</i>	is <i>scalar-int-expr</i>	
18	R1128 <i>concurrent-step</i>	is <i>scalar-int-expr</i>	
19	R1129 <i>concurrent-locality</i>	is [<i>locality-spec</i>]...	
20	R1130 <i>locality-spec</i>	is LOCAL (<i>variable-name-list</i>)	
21		or LOCAL INIT (<i>variable-name-list</i>)	
22		or REDUCE (<i>reduce-operation</i> : <i>variable-name-list</i>)	
23		or SHARED (<i>variable-name-list</i>)	
24		or DEFAULT (NONE)	
25	R1131 <i>reduce-operation</i>	is <i>binary-reduce-op</i>	
26		or <i>function-reduction-name</i>	
27	R1132 <i>binary-reduce-op</i>	is +	
28		or *	
29		or .AND.	
30		or .OR.	
31		or .EQV.	
32		or .NEQV.	
33	C1122 The <i>function-reduction-name</i> shall be the name of the standard intrinsic function IAND, IOR, IOR,		
34	MAX, or MIN.		
35	C1123 (R1125) Any procedure referenced in the <i>scalar-mask-expr</i> , including one referenced by a defined		
36	operation , shall be a pure procedure (15.7).		
37	C1124 (R1126) The <i>index-name</i> shall be a named scalar variable of type integer.		

- C1125 (R1126) A *concurrent-limit* or *concurrent-step* shall not contain a reference to any *index-name* in the *concurrent-control-list* in which it appears.

C1126 A *variable-name* in a *locality-spec* shall be the name of a variable in the innermost executable construct or *scoping unit* that includes the DO CONCURRENT statement.

C1127 A *variable-name* in a *locality-spec* shall not be the same as an *index-name* in the *concurrent-header* of the same DO CONCURRENT statement.

C1128 The name of a variable shall not appear in more than one *variable-name-list*, or more than once in a *variable-name-list*, in a given *concurrent-locality*.

C1129 The DEFAULT (NONE) *locality-spec* shall not appear more than once in a given *concurrent-locality*.

C1130 A *variable-name* that appears in a LOCAL or LOCAL_INIT *locality-spec* shall not have the ALLOCATABLE, INTENT (IN), or OPTIONAL attribute, shall not be of finalizable type, shall not have an allocatable ultimate component, shall not be a nonpointer polymorphic dummy argument, and shall not be a coarray or an assumed-size array. A *variable-name* that is not permitted to appear in a variable definition context shall not appear in a LOCAL or LOCAL_INIT *locality-spec*.

C1131 A *variable-name* that appears in a REDUCE *locality-spec* shall not have the ASYNCHRONOUS, INTENT (IN), OPTIONAL, or VOLATILE attribute, shall not be coindexed, and shall not be an assumed-size array. A *variable-name* that is not permitted to appear in a variable definition context shall not appear in a REDUCE *locality-spec*.

C1132 A *variable-name* that appears in a REDUCE *locality-spec* shall be of intrinsic type suitable for the intrinsic operation or function specified by its *reduce-operation*.

C1133 A variable that is referenced by the *scalar-mask-expr* of a *concurrent-header* or by any *concurrent-limit* or *concurrent-step* in that *concurrent-header* shall not appear in a LOCAL *locality-spec* in the same DO CONCURRENT statement.

C1134 If the *locality-spec* DEFAULT (NONE) appears in a DO CONCURRENT statement, a variable that is a local or *construct* entity of a *scope* containing the DO CONCURRENT construct, and that appears in the block of the construct, shall have its locality explicitly specified by that statement.

R1133 *end-do* is *end-do-stmt*
or *continue-stmt*

R1134 *end-do-stmt* is END DO [*do-construct-name*]

C1135 (R1119) If the *do-stmt* of a *do-construct* specifies a *do-construct-name*, the corresponding *end-do* shall be an *end-do-stmt* specifying the same *do-construct-name*. If the *do-stmt* of a *do-construct* does not specify a *do-construct-name*, the corresponding *end-do* shall not specify a *do-construct-name*.

C1136 (R1119) If the *do-stmt* is a *nonlabel-do-stmt*, the corresponding *end-do* shall be an *end-do-stmt*.

C1137 (R1119) If the *do-stmt* is a *label-do-stmt*, the corresponding *end-do* shall be identified with the same *label*.

11.1.7.3 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 (11.1.7.4.5).

11.1.7.4 Execution of a DO construct

11.1.7.4.1 Loop initiation

1 When the DO statement is executed, the DO construct becomes active. If loop-control is

[,] *do-variable* = *scalar-int-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 10.9). 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

The iteration count is zero whenever:

$$m_1 > m_2 \text{ and } m_3 > 0, \text{ or} \\ m_1 < m_2 \text{ and } m_3 < 0.$$

2 If loop-control is omitted, no iteration count is calculated. The effect is as if a large positive iteration count, impossible to decrement to zero, were established. If loop-control is [,] WHILE (*scalar-logical-expr*), the effect is as if loop-control were omitted and the following statement inserted as the first statement of the block:

IF (.NOT. (*scalar-logical-expr*)) EXIT

3 For a DO CONCURRENT construct, the values of the index variables for the iterations of the construct are determined by the rules in 11.1.7.4.2.

4 At the completion of the execution of the DO statement, the execution cycle begins.

11.1.7.4.2 DO CONCURRENT loop control

1 The concurrent-limit and concurrent-step expressions in the concurrent-control-list are evaluated. These expressions may be evaluated in any order. The set of values that a particular index-name variable assumes is determined as follows.

- (1) The lower bound m_1 , the upper bound m_2 , and the step m_3 are of type integer with the same kind type parameter as the index-name. Their values are established by evaluating the first concurrent-limit, the second concurrent-limit, and the concurrent-step expressions, respectively,

- 1 2 A CYCLE statement belongs to a particular DO construct. If the CYCLE statement contains a DO construct
2 name, it belongs to that DO construct; otherwise, it belongs to the innermost DO construct in which it
3 appears.
- 4 3 Execution of a CYCLE statement that belongs to a DO construct that is not a DO CONCURRENT construct
5 causes immediate progression to step (3) of the execution cycle of the DO construct to which it belongs.
- 6 4 Execution of a CYCLE statement that belongs to a DO CONCURRENT construct completes execution of that
7 iteration of the construct.
- 8 5 In a DO construct, a transfer of control to the *end-do* has the same effect as execution of a CYCLE statement
9 belonging to that construct.

10 11.1.7.4.5 Loop termination

- 11 1 For a DO construct that is not a DO CONCURRENT construct, the loop terminates, and the DO construct
12 becomes inactive, when any of the following occurs.
 - 13 • The iteration count is determined to be zero or the *scalar-logical-expr* is false, when tested during
14 step (1) of the above execution cycle.
 - 15 • An *EXIT statement* that belongs to the DO construct is executed.
 - 16 • An *EXIT* or *CYCLE* statement that belongs to an outer construct and is within the DO construct is
17 executed.
 - 18 • A branch occurs within the DO construct and the *branch target statement* is outside the construct.
 - 19 • A *RETURN statement* within the DO construct is executed.
- 20 2 For a DO CONCURRENT construct, the loop terminates, and the DO construct becomes inactive when all of
21 the iterations have completed execution.
- 22 3 When a DO construct becomes inactive, the DO variable, if any, of the DO construct retains its last defined
23 value.

24 11.1.7.5 Additional semantics for DO CONCURRENT constructs

- 25 C1140 A *RETURN statement* shall not appear within a DO CONCURRENT construct.
- 26 C1141 An *image control statement* shall not appear within a DO CONCURRENT construct.
- 27 C1142 A branch (11.2) within a DO CONCURRENT construct shall not have a branch target that is outside
28 the construct.
- 29 C1143 A reference to an impure procedure shall not appear within a DO CONCURRENT construct.
- 30 C1144 A statement that might result in the deallocation of a *polymorphic* entity shall not appear within a
31 DO CONCURRENT construct.
- 32 C1145 A reference to the procedure *IEEE_GET_FLAG*, *IEEE_GET_HALTING_MODE*, *IEEE_GET_STATUS*,
33 *IEEE_SET_HALTING_MODE*, *IEEE_SET_MODES*, or *IEEE_SET_STATUS* from the intrinsic module
34 *IEEE_EXCEPTIONS*, shall not appear within a DO CONCURRENT construct.
- 35 C1146 A reference to the procedure *IEEE_SET_ROUNDING_MODE* or *IEEE_SET_UNDERFLOW_MODE* from
36 the intrinsic module *IEEE_ARITHMETIC* shall not appear within a DO CONCURRENT construct.

- 1 The locality of a variable that appears in a DO CONCURRENT construct is **LOCAL**, **LOCAL_INIT**, **REDUCE**, **SHARED**, or unspecified. A **construct** or **statement** entity of a construct or statement within the DO CONCURRENT construct has **SHARED** locality if it has the **SAVE attribute**. If it does not have the **SAVE attribute**, it is a different entity in each iteration, similar to **LOCAL** locality.
- 2 A variable that has **LOCAL** or **LOCAL_INIT** locality is a construct entity with the same type, type parameters, and rank as the variable with the same name in the innermost executable construct or **scoping unit** that includes the DO CONCURRENT construct, and the outside variable is inaccessible by that name within the construct. The construct entity has the **ASYNCHRONOUS**, **CONTIGUOUS**, **POINTER**, **TARGET**, or **VOLATILE** attribute if and only if the outside variable has that attribute; it does not have the **BIND**, **INTENT**, **PROTECTED**, **SAVE**, or **VALUE** attribute, even if the outside variable has that attribute. If it is not a pointer, it has the same bounds as the outside variable. At the beginning of execution of each iteration,
- if a variable with **LOCAL** locality is a pointer it has undefined **pointer association** status, and otherwise it is undefined except for any subobjects that are default-initialized;
 - a variable with **LOCAL_INIT** locality has the **pointer association** status and definition status of the outside variable with that name; the outside variable shall not be an undefined pointer or a nonallocatable nonpointer variable that is undefined.
- If a variable with **LOCAL** or **LOCAL_INIT** locality becomes an affector of a pending input/output operation, the operation shall have completed before the end of the iteration. If a variable with **LOCAL** or **LOCAL_INIT** locality has the **TARGET attribute**, a pointer associated with it during an iteration becomes undefined when execution of that iteration completes.
- 3 A variable that has **REDUCE** locality is a construct entity with the same type, type parameters, rank, and bounds as the variable with the same name in the innermost executable construct or scoping unit that includes the DO CONCURRENT construct (the outside variable); the outside variable is inaccessible by that name within the construct. The outside variable shall not be an unallocated allocatable variable or a pointer that is not associated. The construct entity has the **CONTIGUOUS attribute** if and only if the outside variable has that attribute; it does not have the **ALLOCATABLE**, **BIND**, **INTENT**, **POINTER**, **PROTECTED**, **SAVE**, **TARGET**, or **VALUE** attribute, even if the outside variable has that attribute. Before execution of the iterations begins, the construct entity is assigned an initial value corresponding to its *reduce-operation* as specified in Table 11.1.

Table 11.1 — Initial values for reduction operations

Operation	Initial value
+	0
*	1
.AND.	.TRUE.
.OR.	.FALSE.
.EQV.	.TRUE.
.NEQV.	.FALSE.
IAND	All bits set
IEOR	0
IOR	0
MAX	Least representable value of the type and kind
MIN	Largest representable value of the type and kind

NOTE 1

A processor can implement a DO CONCURRENT construct in a manner such that a variable with **REDUCE** locality might not have the initial value from Table 11.1 at the start of every iteration.

- 4 A variable that has **REDUCE** locality shall only appear within the *block* of a DO CONCURRENT construct in the *designator* of a *variable*, as the *object-name*, or as the leftmost *part-name* of an *array-element* or *array-section*, in an *intrinsic assignment statement* with the following forms:

variable = *variable* *binary-reduce-op* *expr*

variable = *expr* *binary-reduce-op* *variable*

variable = *function-reduction-name* ([*expr*,]... *variable* [, *expr*]...)

where each occurrence of *variable* has the same form.

5 If a variable has **REDUCE** locality, on termination of the DO CONCURRENT construct the outside variable is updated by combining it with the values the construct entity had at completion of each iteration, using the *reduce-operation*. The processor may combine the values in any order.

6 If a variable has **SHARED** locality, appearances of the variable within the DO CONCURRENT construct refer to the variable in the innermost executable construct or *scoping unit* that includes the DO CONCURRENT construct. If it is defined or becomes undefined during any iteration, it shall not be referenced, defined, or become undefined during any other iteration. If it is allocated, deallocated, nullified, or pointer-assigned during an iteration it shall not have its allocation or *association* status, dynamic type, array bounds, shape, or a *deferred type parameter* value inquired about in any other iteration. A noncontiguous array with **SHARED** locality shall not be supplied as an *actual argument* corresponding to a *contiguous INTENT (IN-OUT) dummy argument*.

7 If a variable has unspecified locality,

- if it is referenced in an iteration it shall either be previously defined during that iteration, or shall not be defined or become undefined during any other iteration; if it is defined or becomes undefined by more than one iteration it becomes undefined when the loop terminates;
- if it is noncontiguous and is supplied as an *actual argument* corresponding to a *contiguous INTENT (INOUT) dummy argument* in an iteration, it shall either be previously defined in that iteration or shall not be defined in any other iteration;
- if it is a pointer and is used in an iteration other than as the pointer in pointer assignment, allocation, or nullification, it shall either be previously *pointer associated* during that iteration or shall not have its *pointer association* changed during any iteration;
- if it is a pointer whose *pointer association* is changed in more than one iteration, it has an association status of undefined when the construct terminates;
- if it is *allocatable* and is allocated in more than one iteration, it shall have an allocation status of unallocated at the end of every iteration;
- if it is *allocatable* and is referenced, defined, deallocated, or has its allocation status, dynamic type, or a deferred type parameter value inquired about, in any iteration, it shall either be previously allocated in that iteration or shall not be allocated or deallocated in any other iteration.

8 A DO CONCURRENT construct shall not contain an input/output statement that has an **ADVANCE=** specifier.

9 If data are written to a file record or position in one iteration, that record or position in that file shall not be read from or written to in a different iteration. If records are written to a file connected for sequential access by more than one iteration, the ordering of records written by different iterations is processor dependent.

NOTE 2

The restrictions on referencing variables defined in an iteration of a DO CONCURRENT construct apply to any procedure invoked within the loop.

NOTE 3

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.

1 11.1.7.6 Examples of DO constructs

NOTE 1

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 2

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 3

The following example behaves exactly the same as the one in NOTE 2. 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 4

The following example illustrates a case in which the user knows that there are no repeated values in the index array IND. The **DO CONCURRENT** construct makes it easier for the processor to generate vector gather/scatter code, unroll the loop, or parallelize the code for this loop, potentially improving performance.

```
INTEGER :: A(N), IND(N)
...
DO CONCURRENT (I=1:M)
  A(IND(I)) = I
END DO
```

NOTE 5

The following code demonstrates the use of the **LOCAL** clause so that the X inside the **DO CONCURRENT** construct is a temporary variable, and will not affect the X outside the construct.

```
X = 1.0
DO CONCURRENT (I=1:10) LOCAL (X)
  IF (A (I) > 0) THEN
    X = SQRT (A (I))
    A (I) = A (I) - X**2
  END IF
  B (I) = B (I) - A (I)
```

NOTE 5 (cont.)

END DO PRINT *, X	! Always prints 1.0.

NOTE 6

Additional examples of DO constructs are in C.7.3 .

11.1.8 IF construct and statement**11.1.8.1 Purpose and form of the IF construct**

- 1 The IF construct selects for execution at most one of its constituent blocks. The selection is based on a sequence of logical expressions.

```

R1136 if-construct           is  if-then-stmt
                                block
                                [ else-if-stmt
                                block ] ...
                                [ else-stmt
                                block ]
                                end-if-stmt

```

```

R1137 if-then-stmt           is  [ if-construct-name : ] IF ( scalar-logical-expr ) THEN

```

```

R1138 else-if-stmt           is  ELSE IF ( scalar-logical-expr ) THEN [ if-construct-name ]

```

```

R1139 else-stmt             is  ELSE [ if-construct-name ]

```

```

R1140 end-if-stmt           is  END IF [ if-construct-name ]

```

C1147 (R1136) If the *if-then-stmt* of an *if-construct* specifies an *if-construct-name*, the corresponding *end-if-stmt* shall specify the same *if-construct-name*. If the *if-then-stmt* of an *if-construct* does not specify an *if-construct-name*, the corresponding *end-if-stmt* shall not specify an *if-construct-name*. If an *else-if-stmt* or *else-stmt* specifies an *if-construct-name*, the corresponding *if-then-stmt* shall specify the same *if-construct-name*.

11.1.8.2 Execution of an IF construct

- 1 At most one of the blocks in the IF construct is executed. If there is an ELSE statement in the construct, exactly one of the blocks in the construct is executed. The scalar logical expressions are evaluated in the order of their appearance in the construct until a true value is found or an ELSE statement or END IF statement is encountered. If a true value or an ELSE statement is found, the block immediately following is executed and this completes the execution of the construct. The scalar logical expressions in any remaining ELSE IF statements of the IF construct are not evaluated. If none of the evaluated expressions is true and there is no ELSE statement, the execution of the construct is completed without the execution of any block within the construct.
- 2 It is permissible to branch to an END IF statement only from within its IF construct. Execution of an END IF statement has no effect.

11.1.8.3 Examples of IF constructs

C1149 (R1142) 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*.

6 R1146 *case-expr* **is** *scalar-expr*

7 C1150 *case-expr* shall be of type character, integer, or logical, or of enum or enumeration type.

```

8   R1147 case-selector      is ( case-value-range-list )
9                                     or DEFAULT

```

10 C1151 (R1142) No more than one of the selectors of one of the CASE statements shall be DEFAULT.

```

11      R1148 case-value-range           is case-value
12                                     or case-value :
13                                     or : case-value
14                                     or case-value : case-value

```

```

15      R1149 case-value           is scalar-constant-expr

```

C1152 (R1142) For a given *case-construct*, each *case-value* shall be of the same type as *case-expr*, or in type conformance as specified in Table 10.8 if *case-expr* is of an enum type. For character type, the *kind type parameters* shall be the same; character length differences are allowed.

19 C1153 (R1142) A *case-value-range* using a colon shall not be used if *case-expr* is of type logical.

20 C1154 (R1142) For a given *case-construct*, there shall be no possible value of the *case-expr* that matches
21 more than one *case-value-range*.

22 11.1.9.2 Execution of a SELECT CASE construct

23 1 The execution of the SELECT CASE statement causes the case expression to be evaluated. For a case value
24 range list, a match occurs if the case expression value matches any of the case value ranges in the list. For
25 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 other types if the expression $c == v$ is true.

29 (2) If the case value range is of the form $low : high$, a match occurs if the expression $low \leq c$.AND.
30 $c \leq high$ is true.

31 (3) If the case value range is of the form low :, a match occurs if the expression $low \leq c$ is true.

32 (4) If the case value range is of the form : *high*, a match occurs if the expression $c \leq high$ is true.

33 (5) If no other selector matches and a DEFAULT selector appears, it matches the case index.

34 (6) If no other selector matches and the DEFAULT selector does not appear, there is no match.

35 2 The block following the CASE statement containing the matching selector, if any, is executed. This completes
36 execution of the construct.

37 3 It is permissible to branch to an *end-select-stmt* only from within its SELECT CASE construct.

1 **11.1.9.3 Examples of SELECT CASE constructs****NOTE 1**

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 2

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 3

The following three fragments are equivalent:

```

IF (SILLY == 1) THEN      ! Fragment one
    CALL THIS
ELSE
    CALL THAT
END IF

SELECT CASE (SILLY == 1) ! Fragment two
CASE (.TRUE.)
    CALL THIS
CASE (.FALSE.)
    CALL THAT
END SELECT

SELECT CASE (SILLY)      ! Fragment three
CASE DEFAULT
    CALL THAT
CASE (1)
    CALL THIS
END SELECT

```

NOTE 4

A code fragment showing several selections of one block:

NOTE 4 (cont.)

```
SELECT CASE (N)
CASE (1, 3:5, 8)  ! Selects 1, 3, 4, 5, 8
    CALL SUB
CASE DEFAULT
    CALL OTHER
END SELECT
```

11.1.10 SELECT RANK construct

11.1.10.1 Purpose and form of the SELECT RANK construct

- 1 The SELECT RANK construct selects for execution at most one of its constituent blocks. The selection is based on the rank of an [assumed-rank variable](#). A name is associated with the variable ([19.4](#), [19.5.1.6](#)), in the same way as for the [ASSOCIATE construct](#).

```
R1150 select-rank-construct      is    select-rank-stmt
                                   [ select-rank-case-stmt
                                   block ]...
                                   end-select-rank-stmt
```

R1151 *select-rank-stmt* is [*select-construct-name* :] SELECT RANK ■
 ■ ([*associate-name* =>] *selector*)

C1155 The *selector* in a *select-rank-stmt* shall be the name of an *assumed-rank* array.

R1152 *select-rank-case-stmt* **is** RANK (*scalar-int-constant-expr*) [*select-construct-name*]
 or RANK (*) [*select-construct-name*]
 or RANK DEFAULT [*select-construct-name*]

C1156 A *scalar-int-constant-expr* in a *select-rank-case-stmt* shall be nonnegative.

C1157 For a given *select-rank-construct*, the same rank value shall not be specified in more than one *select-rank-case-stmt*.

C1158 For a given *select-rank-construct*, there shall be at most one RANK (*) *select-rank-case-stmt* and at most one RANK DEFAULT *select-rank-case-stmt*.

C1159 If *select-construct-name* appears on a *select-rank-case-stmt*, the corresponding *select-rank-stmt* shall specify the same *select-construct-name*.

C1160 A SELECT RANK construct shall not have a *select-rank-case-stmt* that is RANK (*) if the selector has the **ALLOCATABLE** or **POINTER** attribute.

R1153 *end-select-rank-stmt* **is** END SELECT [*select-construct-name*]

C1161 If the *select-rank-stmt* of a *select-rank-construct* specifies a *select-construct-name*, the corresponding *end-select-rank-stmt* shall specify the same *select-construct-name*. If the *select-rank-stmt* of a *select-rank-construct* does not specify a *select-construct-name*, the corresponding *end-select-rank-stmt* shall not specify a *select-construct-name*.

2 The associate name of a SELECT RANK construct is the *associate-name* if specified; otherwise it is the name that constitutes the selector.

3 The *scalar-int-constant-expr* in a *select-rank-case-stmt* may have a value greater than the maximum possible

rank of the selector; in this case, its block will never be executed.

11.1.10.2 Execution of the SELECT RANK construct

1 A SELECT RANK construct selects at most one block to be executed. During execution of that block, the associate name identifies an entity which is associated (19.5.1.6) with the selector. A RANK (*) statement matches the selector if the selector is argument associated with an [assumed-size array](#). A RANK (*scalar-int-constant-expr*) statement matches the selector if the selector has that rank and is not argument associated with an [assumed-size array](#). A RANK DEFAULT statement matches the selector if no other *select-rank-case-stmt* of the construct matches the selector. If a *select-rank-case-stmt* matches the selector, the block following that statement is executed; otherwise, control is transferred to the *end-select-rank-stmt*.

2 It is permissible to branch to an *end-select-rank-stmt* only from within its SELECT RANK construct.

11.1.10.3 Attributes of a SELECT RANK associate name

1 The associating entity (19.5.5) assumes the [declared type](#) and [type parameters](#) of the selector. It is [polymorphic](#) if and only if the selector is [polymorphic](#).

2 Within the block following a RANK DEFAULT statement, the associating entity is [assumed-rank](#) and has exactly the same attributes as the selector. Within the block following a RANK (*) statement, the associating entity has rank 1 and is [assumed-size](#), as if it were declared with DIMENSION(1:*). Within the block following a RANK (*scalar-int-constant-expr*) statement, the associating entity has the specified rank; the lower bound of each dimension is the result of the intrinsic function LBOUND (16.9.119) applied to the corresponding dimension of the selector, and the upper bound of each dimension is the result of the intrinsic function UBOUND (16.9.215) applied to the corresponding dimension of the selector.

3 The associating entity has the [ALLOCATABLE](#), [POINTER](#), or [TARGET](#) attribute if the selector has that attribute. The other attributes of the associating entity are described in 11.1.3.3.

11.1.10.4 Examples of the SELECT RANK construct

NOTE 1

This example shows how to use a SELECT RANK construct to process scalars and rank-2 arrays; anything else will be rejected as an error.

```
SUBROUTINE process(x)
  REAL x(..)
  !
  SELECT RANK(x)
  RANK (0)
    x = 0
  RANK (2)
    IF (SIZE(x,2)>=2) x(:,2) = 2
  RANK DEFAULT
    Print *, 'I did not expect rank', RANK(x), 'shape', SHAPE(x)
    ERROR STOP 'process bad arg'
  END SELECT
```

NOTE 2

The following example shows how to process [assumed-size arrays](#), including how to use [sequence association](#) for multi-dimensional processing of an [assumed-size array](#).

```
SELECT RANK (y => x)
RANK (*)
  IF (RANK(x)==2) THEN
    ! Special code for the rank two case.
```

NOTE 2 (cont.)

```

        CALL sequence_assoc_2(y, LBOUND(x,1), UBOUND(x,1), LBOUND(x,2))
    ELSE
        ! We just do all the other ranks in array element order.
        i = 1
        DO
            IF (y(i)==0) Exit
            y(i) = -y(i)
            i = i + 1
        END DO
    END IF
END SELECT
...
CONTAINS
...
SUBROUTINE sequence_assoc_2(a, lb1, ub1, lb2)
    INTEGER, INTENT (IN) :: lb1, ub1, lb2
    REAL a(lb1:ub1,lb2:*)
    j = lb2
outer: DO
    DO i=lb1,ub1
        IF (a(i,j)==0) EXIT outer
        a(i,j) = a(i,j)**2
    END DO
    j = j + 1
    IF (ANY(a(:,j)==0)) EXIT
    j = j + 1
END DO outer
END SUBROUTINE

```

11.1.11 SELECT TYPE construct**11.1.11.1 Purpose and form of the SELECT TYPE construct**

- 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 ([19.4](#), [19.5.1.6](#)), in the same way as for the ASSOCIATE construct.

R1154 *select-type-construct* **is** *select-type-stmt*
 [*type-guard-stmt*
 block] ...
 end-select-type-stmt

R1155 *select-type-stmt* **is** [*select-construct-name* :] SELECT TYPE ■
 ■ ([*associate-name* =>] *selector*)

C1162 (R1155) If *selector* is not a named [variable](#), *associate-name* => shall appear.

C1163 (R1155) If *selector* is not a [variable](#) or is a [variable](#) that has a [vector subscript](#), neither *associate-name* nor any subobject thereof shall appear in a variable definition context ([19.6.7](#)) or pointer association context ([19.6.8](#)).

C1164 (R1155) The *selector* in a *select-type-stmt* shall be [polymorphic](#).

R1156 *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*]

C1165 (R1156) The *type-spec* or *derived-type-spec* shall specify that each length type parameter is assumed.

C1166 (R1156) The *type-spec* or *derived-type-spec* shall not specify a derived type with the *BIND attribute* or the *SEQUENCE attribute*.

C1167 (R1154) 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*.

C1168 (R1154) 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*.

C1169 (R1154) For a given *select-type-construct*, there shall be at most one CLASS DEFAULT *type-guard-stmt*.

R1157 *end-select-type-stmt* **is** END SELECT [*select-construct-name*]

C1170 (R1154) 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*.

11.1.11.2 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 (19.5.1.6) 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 will inevitably specify a type that is an *extension* of all the types specified in the others; the block following that statement is executed.
- (4) Otherwise, if there is a CLASS DEFAULT type guard statement, the block following that statement is executed.
- (5) Otherwise, no block is executed.

NOTE 1

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.

- 1 5 Within the block following a TYPE IS type guard statement, the [associating entity](#) (19.5.5) is not poly-
- 2 morphic (7.3.2.3), has the type named in the type guard statement, and has the type parameter values
- 3 of the selector.
- 4 6 Within the block following a CLASS IS type guard statement, the [associating entity](#) is [polymorphic](#) and has
- 5 the [declared type](#) named in the type guard statement. The type parameter values of the [associating entity](#)
- 6 are the corresponding type parameter values of the selector.
- 7 7 Within the block following a CLASS DEFAULT type guard statement, the [associating entity](#) is [polymorphic](#)
- 8 and has the same [declared type](#) as the selector. The type parameter values of the [associating entity](#) are
- 9 those of the [declared type](#) of the selector.

NOTE 2

If the [declared type](#) of the [selector](#) is T, specifying CLASS DEFAULT has the same effect as specifying CLASS IS (T).

- 10 8 The other attributes of the [associating entity](#) are described in 11.1.3.3.
- 11 9 It is permissible to branch to an [end-select-type-stmt](#) only from within its SELECT TYPE construct.

12 **11.1.11.3 Examples of the SELECT TYPE construct****NOTE 1**

```

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 2

The following example illustrates the omission of *associate-name*. It uses the declarations from NOTE 1.

```

P_OR_C => P3
SELECT TYPE ( P_OR_C )
CLASS IS ( POINT )
  ! "CLASS ( POINT ) :: P_OR_C" implied here
  PRINT *, P_OR_C%X, P_OR_C%Y
TYPE IS ( POINT_3D )
  ! "TYPE ( POINT_3D ) :: P_OR_C" implied here

```

NOTE 2 (cont.)

```
PRINT *, P_OR_C%X, P_OR_C%Y, P_OR_C%Z  ! This block gets executed
END SELECT
```

11.1.12 EXIT statement

1 The EXIT statement provides one way of terminating a loop, or completing execution of another construct.

R1158 *exit-stmt* is EXIT [*construct-name*]

C1171 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.

C1172 An *exit-stmt* shall not appear within a *DO CONCURRENT construct* if it belongs to that construct or an outer construct.

C1173 An *exit-stmt* shall not appear within a *CHANGE TEAM* or *CRITICAL* construct if it belongs to an outer construct.

3 When an EXIT statement that belongs to a *DO construct* is executed, it terminates the loop (11.1.7.4.5) and any active loops contained within the terminated loop. When an EXIT statement that belongs to a non-DO construct is executed, it terminates any active loops contained within that construct, and completes execution of that construct. If the EXIT statement belongs to a *CHANGE TEAM construct*, the effect is the same as transferring control to the *END TEAM statement*; if that statement contains a *STAT=* or *ERRMSG=* specifier, the *stat-variable* or *errmsg-variable* becomes defined as specified for that statement.

11.2 Branching

11.2.1 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 can 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-rank-stmt*, *end-select-rank-stmt*, *select-type-stmt*, *end-select-type-stmt*, *do-stmt*, *end-do-stmt*, *block-stmt*, *end-block-stmt*, *critical-stmt*, *end-critical-stmt*, *forall-construct-stmt*, *forall-stmt*, *where-construct-stmt*, *end-function-stmt*, *end-mp-subprogram-stmt*, *end-program-stmt*, or *end-subroutine-stmt*.

11.2.2 GO TO statement

R1159 *goto-stmt* is GO TO *label*

C1174 (R1159) 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.

11.2.3 Computed GO TO statement

1 The computed GO TO statement is obsolescent.

R1160 *computed-goto-stmt* is GO TO (*label-list*) [,] *scalar-int-expr*

C1175 (R1160) 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*.

2 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.

11.3 CONTINUE statement

1 Execution of a CONTINUE statement has no effect.

R1161 *continue-stmt* is CONTINUE

11.4 STOP and ERROR STOP statements

R1162 *stop-stmt* is STOP [*stop-code*] [, QUIET = *scalar-logical-expr*]

R1163 *error-stop-stmt* is ERROR STOP [*stop-code*] [, QUIET = *scalar-logical-expr*]

R1164 *stop-code* is *scalar-default-char-expr*
or *scalar-int-expr*

C1176 (R1164) The *scalar-int-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 the *stop-code* is an integer, it is recommended that the value be used as the process exit status, if the processor supports that concept. If the *stop-code* in a STOP statement is of type character or does not appear, or if an *end-program-stmt* is executed, it is recommended that the value zero be supplied as the process exit status, if the processor supports that concept. If the *stop-code* in an ERROR STOP statement is of type character or does not appear, it is recommended that a processor-dependent nonzero value be supplied as the process exit status, if the processor supports that concept.

3 If QUIET= is omitted or the *scalar-logical-expr* has the value false:

- if any exception (17) is signaling on that *image*, the processor shall issue a warning indicating which exceptions are signaling, and this warning shall be on the *unit* identified by the *named constant ERROR_UNIT* from the intrinsic module *ISO_FORTRAN_ENV* (16.10.2.9);
- if a stop code is specified, it is recommended that it be made available by formatted output to the same *unit*.

4 If QUIET= appears and the *scalar-logical-expr* has the value true, no output of signaling exceptions or stop code shall be produced.

NOTE 1

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 2

If the integer [stop-code](#) is used as the process exit status, the processor might be able to interpret only values within a limited range, or only a limited portion of the integer value (for example, only the least-significant 8 bits).

11.5 FAIL IMAGE statement

R1165 *fail-image-stmt* **is** FAIL IMAGE

- 1 Execution of a FAIL IMAGE statement causes the executing image to cease participating in program execution without initiating termination. No further statements are executed by that image.

NOTE

The FAIL IMAGE statement enables testing of a recovery algorithm without needing an actual failure.

On a processor that does not have the ability to detect that an image has failed, execution of a FAIL IMAGE statement might provide a simulated failure environment that provides debug information.

In a piece of code that executes about once a second, invoking this subroutine on an image

```
SUBROUTINE FAIL
  REAL :: X
  CALL RANDOM_NUMBER (X)
  IF (X<0.001) FAIL IMAGE
END SUBROUTINE FAIL
```

will cause that image to have approximately a 1/1000 chance of failure every second.

Note that FAIL IMAGE is not an [image control statement](#).

11.6 NOTIFY WAIT statement

- 1 The NOTIFY WAIT statement waits until the value of its [notify-variable](#) is greater than or equal to a threshold value.

R1166 *notify-wait-stmt* **is** NOTIFY WAIT ([notify-variable](#) [, [event-wait-spec-list](#)])

R1167 *notify-variable* **is** [scalar-variable](#)

C1177 A [notify-variable](#) shall be of type NOTIFY_TYPE from the intrinsic module ISO_FORTRAN_ENV.

C1178 A [notify-variable](#) shall not be a [coindexed object](#).

- 2 The [notify-variable](#) shall not depend on the value of [stat-variable](#) or [errmsg-variable](#).

- 3 Execution of a NOTIFY WAIT statement consists of the following sequence of actions:

- (1) if the UNTIL_COUNT= specifier appears and its [scalar-int-expr](#) is greater than one, the threshold value is set to that value, otherwise, the threshold value is set to one;
- (2) the executing image waits until the count of the notify variable is greater than or equal to the threshold value or an error condition occurs;
- (3) if no error condition occurs, the count of the notify variable is atomically decremented by the threshold value.

- 1 4 If an error condition occurs during execution of a NOTIFY WAIT statement, the value of the count of its
2 notify variable is processor dependent.
- 3 5 Execution of an [assignment statement](#) whose variable has a [NOTIFY= specifier](#) is initially unsatisfied. Suc-
4 cessful execution of a NOTIFY WAIT statement with a threshold value of k satisfies the first k unsatisfied
5 executions of [assignment statements](#) whose [NOTIFY= specifier](#) specifies the same notify variable as the
6 NOTIFY WAIT statement.
- 7 6 The [stat-variable](#) of a NOTIFY WAIT statement shall not depend on the value of the notify variable or the
8 [errmsg-variable](#). The [errmsg-variable](#) of a NOTIFY WAIT statement shall not depend on the value of the
9 notify variable or the [stat-variable](#).
- 10 7 If a NOTIFY WAIT statement has a [STAT= specifier](#), [stat-variable](#) is assigned the value zero if execution
11 of the statement is successful, and a processor-dependent positive value that is different from the value
12 of [STAT_FAILED_IMAGE](#) (16.10.2.28) and [STAT_STOPPED_IMAGE](#) (16.10.2.31) from the intrinsic module
13 [ISO_FORTRAN_ENV](#) (16.10.2) if an error condition occurs.
- 14 8 If an error condition occurs during execution of a NOTIFY WAIT statement with no [STAT=](#), [error termination](#)
15 is initiated.
- 16 9 If a NOTIFY WAIT statement has an [ERRMSG= specifier](#) and an error condition occurs, [errmsg-variable](#) is
17 assigned an explanatory message, as if by intrinsic assignment. If no such condition occurs, the definition
18 status and the value of [errmsg-variable](#) are unchanged.
- 19 10 The set of error conditions that can occur during execution of a NOTIFY WAIT statement is processor de-
20 pendent.

21 11.7 Image execution control

22 11.7.1 Image control statements

- 23 1 The execution sequence on each [image](#) is specified in 5.3.5.
- 24 2 Execution of an [image control statement](#) divides the execution sequence on an [image](#) into [segments](#). Each
25 of the following is an [image control statement](#):
 - 26 • [SYNC ALL statement](#);
 - 27 • [SYNC IMAGES statement](#);
 - 28 • [SYNC MEMORY statement](#);
 - 29 • [SYNC TEAM statement](#);
 - 30 • [ALLOCATE statement](#) that has a [coarray allocate-object](#);
 - 31 • [DEALLOCATE statement](#) that has an [allocate-object](#) that is a [coarray](#) or has a [coarray potential sub-](#)
32 [object component](#);
 - 33 • [CHANGE TEAM](#) or [END TEAM](#) statement (11.1.5);
 - 34 • [CRITICAL](#) or [END CRITICAL](#) statement (11.1.6);
 - 35 • [EVENT POST](#) or [EVENT WAIT](#) statement;
 - 36 • [FORM TEAM statement](#);
 - 37 • [LOCK](#) or [UNLOCK](#) statement;
 - 38 • 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 During an execution of a statement that invokes more than one procedure, at most one invocation shall cause execution of an **image control statement** other than **CRITICAL** or **END CRITICAL**.

11.7.2 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. If an **image** does not execute any **image control statement** before **termination of execution**, its entire statement execution sequence is a single **segment**.

2 By execution of **image control statements** or user-defined ordering (11.7.5), 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**. An **event variable** or **notify variable** may be referenced or defined during the execution of a **segment** that is unordered relative to the execution of another **segment** in which that **event variable** or **notify variable** is defined. A variable defined in an unordered **segment** only by execution of an **assignment statement** with a **NOTIFY= specifier** may be referenced or defined after execution of a **NOTIFY WAIT statement** that satisfies that **assignment statement** execution. Otherwise,

- if a variable is defined or becomes undefined on an **image** in a **segment**, it shall not be referenced, defined, or become undefined in a **segment** on another **image** unless the **segments** are ordered,
- if the allocation of an **allocatable** subobject of a **coarray** or the **pointer association** of a pointer subobject of a **coarray** is changed on an **image** in a **segment**, that subobject shall not be referenced, defined, or have its allocation or **association** status, **dynamic type**, array **bounds**, **shape**, or a **deferred type parameter** value inquired about in a **segment** on another **image** unless the **segments** are ordered, and
- if a procedure invocation on **image** P is in execution in **segments** $P_i, P_{i+1}, \dots, P_k$ 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 .

4 If, by execution of a statement in **segment** P_i on **image** P,

- a variable X is defined, referenced, becomes undefined, or has its allocation status, **pointer association** status, array **bounds**, **dynamic type**, or **type parameters** changed or inquired about,
- **segment** P_i on **image** P precedes **segment** Q_j on **image** Q, and
- X 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 in **segment** Q_j on **image** Q,

- 1 then the action regarding X in **segment** P_i on image P precedes the action regarding X in **segment** Q_j on
 2 image Q.

NOTE 1

The set of all **segments** on all **images** is partially ordered: the **segment** P_i precedes **segment** Q_j if and only if there is a sequence of **segments** starting with P_i and ending with Q_j such that each **segment** of the sequence precedes the next either because they are consecutive **segments** on the same **image** or because of the execution of **image control statements**.

NOTE 2

If the **segments** $S_1, S_2, \dots, S_k$ on the distinct **images** $P_1, P_2, \dots, P_k$ 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 3

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 4

The model upon which the interpretation of a program is based is that there is a permanent memory location for each **coarray** and that all **images** on which it is established can access it.

In practice, apart from executions of **atomic subroutines**, the processor could make a copy of a nonvolatile **coarray** in a **segment** (in cache or a register, for example) and, as an optimization, defer copying a changed value back to its permanent memory location while it is still being used. Since the variable is not volatile, it is safe to defer this copying back until the end of the **segment**. It might not be safe to defer this action 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 5

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**.

3 **11.7.3 SYNC ALL statement**

4 R1168 *sync-all-stmt* **is** SYNC ALL [([*sync-stat-list*])]

5 R1169 *sync-stat* **is** STAT = *stat-variable*
 6 **or** ERRMSG = *errmsg-variable*

7 C1179 No specifier shall appear more than once in a given *sync-stat-list*.

8 C1180 A *stat-variable* or *errmsg-variable* in a *sync-stat* shall not be a **coindexed object**.

9 1 The STAT= and ERRMSG= specifiers for **image control statements** are described in 11.7.11.

10 2 Successful execution of a SYNC ALL statement performs a synchronization of all **images** in the **current team**.
 11 Execution on an **image**, M, of the **segment** following the SYNC ALL statement is delayed until each other
 12 **image** in the **current team** has executed a SYNC ALL statement as many times as has **image** M in this **team**.

- 1 The **segments** that executed before the SYNC ALL statement on an **image** precede the **segments** that execute
 2 after the SYNC ALL statement on another **image**.

NOTE

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[*]
...
SYNC ALL
IF (THIS_IMAGE()==1) THEN
  READ (*,*) P
  DO I = 2, NUM_IMAGES()
    P[I] = P
  END DO
END IF
SYNC ALL

```

3 11.7.4 SYNC IMAGES statement

4 R1170 *sync-images-stmt* is SYNC IMAGES (*image-set* [, *sync-stat-list*])

5 R1171 *image-set* is *int-expr*
 6 or *

7 C1181 An *image-set* that is an *int-expr* shall be scalar or of **rank** one.

8 C1182 The value of *image-set* shall not depend on the value of *stat-variable* or *errmsg-variable*.

- 9 1 If *image-set* is an array expression, the value of each element shall be positive and not greater than the
 10 number of **images** in the **current team**, and there shall be no repeated values.
- 11 2 If *image-set* is a scalar expression, its value shall be positive and not greater than the number of **images** in
 12 the **current team**.
- 13 3 An *image-set* that is an asterisk specifies all **images** in the **current team**.
- 14 4 Execution of a SYNC IMAGES statement performs a synchronization of the **image** with each of the other
 15 **images** in the *image-set*. Executions of SYNC IMAGES statements on **images** M and T correspond if the
 16 number of times **image** M has executed a SYNC IMAGES statement in the **current team** with T in its **image**
 17 set is the same as the number of times **image** T has executed a SYNC IMAGES statement with M in its **image**
 18 set in this **team**. The **segments** that executed before the SYNC IMAGES statement on either **image** precede
 19 the **segments** that execute after the corresponding SYNC IMAGES statement on the other **image**.

NOTE 1

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 2

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

NOTE 2 (cont.)

execute the statement SYNC IMAGES (1). The other **images** wait for **image** 1 to set up the data, but do not wait on any other **image**.

```
IF (THIS_IMAGE () == 1) then
  ! Set up coarray data needed by all other images.
  SYNC IMAGES (*)
ELSE
  SYNC IMAGES (1)
  ! Use the data set up by image 1.
END IF
```

When the following example runs on five or more **images**, each **image** synchronizes with both of its neighbors, in a circular fashion.

```
INTEGER :: up, down
...
IF (NUM_IMAGES () > 1) THEN
  up = THIS_IMAGE () + 1; IF (up>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 3

In the following example, each **image** synchronizes with its neighbor.

```
INTEGER :: ME, NE, STEP, NSTEPS
NE = NUM_IMAGES ()
ME = THIS_IMAGE ()
... ! Initial calculation
SYNC ALL
DO STEP = 1, NSTEPS
  IF (ME > 1) SYNC IMAGES (ME-1)
  ... ! Perform calculation
  IF (ME < NE) SYNC IMAGES (ME+1)
END DO
SYNC ALL
```

The calculation starts on **image** 1 since all the others will be waiting on SYNC IMAGES (ME-1). When this is done, **image** 2 can start and **image** 1 can perform its second calculation. This continues until they are all executing different steps at the same time. Eventually, **image** 1 will finish and then the others will finish one by one.

11.7.5 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**.

R1172 *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,

- 1 then the action regarding X on **image** Q precedes the action regarding Y on **image** Q.
- 2 3 User-defined ordering of **segment** P_i on **image** P to precede **segment** Q_j on **image** Q occurs when
- 3 • **image** P executes an **image control statement** that ends **segment** P_i , and then executes statements
 - 4 that initiate a cooperative synchronization between **images** P and Q, and
 - 5 • **image** Q executes statements that complete the cooperative synchronization between **images** P and
 - 6 Q and then executes an **image control statement** that begins **segment** Q_j .
- 7 4 Execution of the cooperative synchronization between **images** P and Q shall include a dependency that
- 8 forces execution on **image** P of the statements that initiate the synchronization to precede the execution on
- 9 **image** Q of the statements that complete the synchronization. The mechanisms available for creating such
- 10 a dependency are processor dependent.

NOTE 1

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 2

SYNC MEMORY can be used to implement specialized schemes for **segment** ordering. For example, the user might have access to an **external procedure** that performs synchronization between **images**. That library procedure might not be aware of the mechanisms used by the processor to manage remote data references and definitions, and therefore not, by itself, be able to ensure the correct memory state before and after its reference. The SYNC MEMORY statement provides the needed memory ordering that enables the safe use of the external synchronization routine. For example:

```

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**.

11.7.6 SYNC TEAM statement

R1173 *sync-team-stmt* is SYNC TEAM (*team-value* [, *sync-stat-list*])

- 1 The *team-value* shall identify an **ancestor team**, the **current team**, or a **team** whose **parent** is the **current team**. The executing **image** shall be a member of the specified **team**.
- 2 Successful execution of a SYNC TEAM statement performs a synchronization of the team identified by *team-value*. Execution on an image, M, of the **segment** following the SYNC TEAM statement is delayed until each other image of the specified team has executed a SYNC TEAM statement specifying the same team as many times as has image M in this team. The **segments** that executed before the SYNC TEAM statement on an image precede the **segments** that execute after the corresponding SYNC TEAM statement on another image.

NOTE

A SYNC TEAM statement synchronizes a particular team whereas a SYNC ALL statement synchronizes the current team.

11.7.7 EVENT POST statement

1 The EVENT POST statement posts an event.

R1174 *event-post-stmt* **is** EVENT POST (*event-variable* [, *sync-stat-list*])

R1175 *event-variable* **is** *scalar-variable*

C1183 (R1175) An *event-variable* shall be of type **EVENT_TYPE** from the intrinsic module **ISO_FORTRAN_ENV** (16.10.2).

2 The *event-variable* shall not depend on the value of *stat-variable* or *errmsg-variable*.

3 Successful execution of an EVENT POST statement atomically increments the count of the *event variable* by one. If an error condition occurs during execution of an EVENT POST statement, the value of the count of the *event variable* is processor dependent. The completion of an EVENT POST statement does not depend on the execution of a corresponding **EVENT WAIT statement**.

11.7.8 EVENT WAIT statement

1 The EVENT WAIT statement waits until an event is posted.

R1176 *event-wait-stmt* **is** EVENT WAIT (*event-variable* [, *event-wait-spec-list*])

R1177 *event-wait-spec* **is** *until-spec*
or *sync-stat*

R1178 *until-spec* **is** UNTIL_COUNT = *scalar-int-expr*

C1184 (R1176) The *event-variable* in an *event-wait-stmt* shall not be *coindexed*.

C1185 No specifier shall appear more than once in a given *event-wait-spec-list*.

2 The *event-variable* shall not depend on the value of *stat-variable* or *errmsg-variable*.

3 Execution of an EVENT WAIT statement consists of the following sequence of actions:

1. if the UNTIL_COUNT= specifier does not appear, the threshold value is set to one; otherwise, the threshold value is set to the maximum of the value of the *scalar-int-expr* and one;
2. the executing image waits until the count of the *event variable* is greater than or equal to the threshold value or an error condition occurs;
3. if no error condition occurs, the count of the *event variable* is atomically decremented by the threshold value.

4 If an error condition occurs during execution of an EVENT WAIT statement, the value of the count of its *event variable* is processor dependent.

5 An **EVENT POST statement** execution is initially unsatisfied. Successful execution of an EVENT WAIT statement with a threshold of *k* satisfies the first *k* unsatisfied **EVENT POST statement** executions for that *event*

variable. This EVENT WAIT statement execution causes the **segment** following the EVENT WAIT statement execution to succeed the **segments** preceding those **k EVENT POST statement** executions.

11.7.9 FORM TEAM statement

1 The FORM TEAM statement creates a set of **sibling teams** whose **parent team** is the **current team**.

R1179 *form-team-stmt* **is** FORM TEAM (*team-number*, *team-variable* ■
 ■ [, *form-team-spec-list*])

R1180 *team-number* **is** *scalar-int-expr*

R1181 *team-variable* **is** *scalar-variable*

C1186 A *team-variable* shall be of type **TEAM_TYPE** from the intrinsic module **ISO_FORTRAN_ENV**.

R1182 *form-team-spec* **is** NEW_INDEX = *scalar-int-expr*
 or *sync-stat*

C1187 No specifier shall appear more than once in a given *form-team-spec-list*.

2 Successful execution of a FORM TEAM statement creates a new team for each unique *team-number* value specified by the **active images** of the **current team**. The value of *team-number* shall be positive. Each executing image will belong to the team whose **team number** is equal to the value of *team-number* on that image, and the *team-variable* becomes defined with a value that identifies that team.

3 The value of the *scalar-int-expr* in a NEW_INDEX= specifier specifies the **image index** that the executing image will have in its new **team**. It shall be positive, less than or equal to the number of **images** in the **team**, and different from the value specified by every other **image** that belongs to that **team**.

4 If the NEW_INDEX= specifier does not appear, the **image index** of the executing image in the new **team** is processor dependent. This **image index** will be positive, less than or equal to the number of **images** in the **team**, and different from that of every other **image** in the **team**.

5 If the FORM TEAM statement is executed on one **image**, the same statement shall be executed on all **active images** of the **current team**. When a FORM TEAM statement is executed, there is an implicit synchronization of all **active images** in the **current team**. On those **images**, execution of the **segment** following the statement is delayed until all other **active images** in the **current team** have executed the same statement the same number of times in this team. The **segments** that executed before the FORM TEAM statement on an **active image** of this **team** precede the **segments** that execute after the FORM TEAM statement on another **active image** of this **team**. If an error condition other than detection of a failed image occurs, the team variable becomes undefined.

6 If execution of a FORM TEAM statement assigns the value **STAT_FAILED_IMAGE** to the *stat-variable*, the effect is the same as for the successful execution of FORM TEAM except for the value assigned to *stat-variable*.

NOTE 1

Executing the statement

```
FORM TEAM ( 2 - MOD (THIS_IMAGE (), 2), ODD_EVEN )
```

will create two subteams of the **current team**, with **images** whose image index is odd being in the team with number 1, and those with an even image index being in the team with number 2.

NOTE 2

If the current team consists of P^2 images, with corresponding coarrays on each image representing parts of a larger array spread over a $P \times P$ square, the following code will establish teams for the rows with image indices equal to the column indices.

```
USE, INTRINSIC :: ISO_FORTRAN_ENV
TYPE (TEAM_TYPE) :: ROW
REAL :: A [P, *]
INTEGER :: ME (2)
ME (:) = THIS_IMAGE (A)
FORM TEAM (ME(1), ROW, NEW_INDEX=ME(2))
```

11.7.10 LOCK and UNLOCK statements

R1183 *lock-stmt* **is** LOCK (*lock-variable* [, *lock-stat-list*])

R1184 *lock-stat* **is** ACQUIRED_LOCK = *scalar-logical-variable*
or *sync-stat*

C1188 No specifier shall appear more than once in a given *lock-stat-list*.

R1185 *unlock-stmt* **is** UNLOCK (*lock-variable* [, *sync-stat-list*])

R1186 *lock-variable* **is** *scalar-variable*

C1189 (R1186) A *lock-variable* shall be of type LOCK_TYPE from the intrinsic module ISO_FORTRAN_ENV (16.10.2.19).

1 The *lock-variable* shall not depend on the value of *stat-variable*, *errmsg-variable*, or the *scalar-logical-variable* in the ACQUIRED_LOCK= specifier. The *scalar-logical-variable* shall not depend on the value of the *lock-variable*, *stat-variable*, or *errmsg-variable*.

2 A lock variable is unlocked if and only if the value of each component is the same as its default value. If it has any other value, it is locked. A lock variable is locked by an *image* if it was locked by execution of a LOCK statement on that *image*, has not been subsequently unlocked by execution of an UNLOCK statement on the same *image*, and that *image* has not failed.

3 Successful execution of a LOCK statement without an ACQUIRED_LOCK= specifier causes the lock variable to become locked by that *image*. If the lock variable is already locked by another *image*, that LOCK statement causes the lock variable to become locked after the other *image* causes the lock variable to become unlocked.

4 If the lock variable is unlocked, successful execution of a LOCK statement with an ACQUIRED_LOCK= specifier causes the lock variable to become locked by that *image* and the scalar logical variable to become defined with the value true. If the lock variable is already locked by a different *image*, successful execution of a LOCK statement with an ACQUIRED_LOCK= specifier leaves the lock variable unchanged and causes the scalar logical variable to become defined with the value false.

5 Successful execution of an UNLOCK statement causes the lock variable to become unlocked. Failure of an *image* causes all lock variables that are locked by that *image* to become unlocked.

6 During execution of the program, the value of a lock variable changes through a sequence of locked and unlocked states due to the execution of LOCK and UNLOCK statements, and by *failure of an image* while it is locked by that *image*. If a lock variable becomes unlocked by execution of an UNLOCK statement on *image* M and next becomes locked by execution of a LOCK statement on *image* T, the *segments* preceding the UNLOCK

- 1 statement on **image** M precede the **segments** following the LOCK statement on **image** T. Execution of a LOCK
 2 statement that does not cause the lock variable to become locked does not affect **segment** ordering.
- 3 7 An error condition occurs if the lock variable in a LOCK statement is already locked by the executing **image**.
 4 An error condition occurs if the lock variable in an UNLOCK statement is not already locked by the executing
 5 **image**. If an error condition occurs during execution of a LOCK or UNLOCK statement, the value of the lock
 6 variable is not changed and the value of the ACQUIRED_LOCK variable, if any, is not changed.

NOTE 1

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 2

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 3

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

```

7 11.7.11 STAT= and ERRMSG= specifiers in image control statements

- 8 1 In an **image control statement**, the **stat-variable** in a **sync-stat** shall not depend on the value of an **errmsg-**
 9 **variable** in a **sync-stat**, **event-variable**, **lock-variable**, **team-variable**, or the **scalar-logical-variable** in the AC-
 10 QUIRED_LOCK= specifier. The **errmsg-variable** in a **sync-stat** shall not depend on the value of a **stat-variable**

- 1 in a *sync-stat*, *event-variable*, *lock-variable*, *team-variable*, or the *scalar-logical-variable* in the ACQUIRED_
2 LOCK= specifier.
- 3 2 If a STAT= specifier appears in a *sync-stat* in an *image control statement*, the *stat-variable* is assigned the
4 value zero if execution of the statement is successful.
- 5 3 If the STAT= specifier appears in a *sync-stat* in an *EVENT WAIT* or *SYNC MEMORY* statement and an er-
6 ror condition occurs, *stat-variable* is assigned a processor-dependent positive value that is different from
7 the value of *STAT_FAILED_IMAGE* (16.10.2.28) and *STAT_STOPPED_IMAGE* (16.10.2.31) from the intrinsic
8 module *ISO_FORTRAN_ENV* (16.10.2).
- 9 4 The *images* involved in execution of an *END TEAM*, *FORM TEAM*, or *SYNC ALL* statement are those in the
10 *current team*. The *images* involved in execution of a *CHANGE TEAM* or *SYNC TEAM* statement are those of
11 the specified *team*. The *images* involved in execution of a *SYNC IMAGES statement* are the *images* specified
12 and the executing *image*. The *images* involved in execution of an *EVENT POST statement* are the *image* on
13 which the *event variable* is located and the executing *image*.
- 14 5 If the STAT= specifier appears in a *sync-stat* in a *CHANGE TEAM*, *END TEAM*, *EVENT POST*, *FORM TEAM*,
15 *SYNC ALL*, *SYNC IMAGES*, or *SYNC TEAM* statement,
- 16 • if one of the *images* involved has stopped, *stat-variable* is assigned the value *STAT_STOPPED_IMAGE*
17 (*16.10.2.31*) from the intrinsic module *ISO_FORTRAN_ENV*;
 - 18 • otherwise, if one of the *images* involved has failed and no other error condition occurs, the inten-
19 ded action is performed on the *active images* involved and *stat-variable* is assigned the value *STAT_-*
20 *FAILED_IMAGE* (*16.10.2.28*) from the intrinsic module *ISO_FORTRAN_ENV*;
 - 21 • otherwise, if any other error condition occurs, *stat-variable* is assigned a processor-dependent pos-
22 itive value that is different from the values of *STAT_STOPPED_IMAGE* and *STAT_FAILED_IMAGE*.
- 23 6 If the STAT= specifier appears in a *sync-stat* in a *SYNC ALL*, *SYNC IMAGES*, or *SYNC TEAM* statement and the
24 error condition *STAT_STOPPED_IMAGE* occurs, the effect is the same as that of executing the *SYNC MEMORY*
25 *statement*, except for defining the *stat-variable*.
- 26 7 If the STAT= specifier appears in a *sync-stat* in a *LOCK statement*,
- 27 • if the *image* on which the *lock variable* is located has failed, the *stat-variable* becomes defined with
28 the value *STAT_FAILED_IMAGE*;
 - 29 • otherwise, if the *lock variable* is locked by the executing *image*, the *stat-variable* becomes defined
30 with the value of *STAT_LOCKED* (*16.10.2.29*) from the intrinsic module *ISO_FORTRAN_ENV*;
 - 31 • otherwise, if the *lock variable* is unlocked because of the failure of the image that locked it, *stat-*
32 *variable* becomes defined with the value *STAT_UNLOCKED_FAILED_IMAGE* (*16.10.2.33*) from the in-
33 trinsic module *ISO_FORTRAN_ENV*.
- 34 8 If the STAT= specifier appears in a *sync-stat* in an *UNLOCK statement*,
- 35 • if the *image* on which the *lock variable* is located has failed, the *stat-variable* becomes defined with
36 the value *STAT_FAILED_IMAGE*;
 - 37 • otherwise, if the *lock variable* has the value unlocked, the *stat-variable* becomes defined with the
38 value of *STAT_UNLOCKED* (*16.10.2.32*) from the intrinsic module *ISO_FORTRAN_ENV*;
 - 39 • otherwise, if the *lock variable* is locked by a different *image*, the *stat-variable* becomes defined with
40 the value *STAT_LOCKED_OTHER_IMAGE* (*16.10.2.30*) from the intrinsic module *ISO_FORTRAN_ENV*.

- 1 9 If the STAT= specifier appears in a *sync-stat* in a **LOCK** or **UNLOCK** statement and any other error condition
2 occurs during execution of that statement, the *stat-variable* becomes defined with a processor-dependent
3 positive value that is different from **STAT_LOCKED**, **STAT_LOCKED_OTHER_IMAGE**, **STAT_UNLOCKED**, and
4 **STAT_UNLOCKED_FAILED_IMAGE**.
- 5 10 If an image completes execution of a **CRITICAL statement** that has a *sync-stat* that is a STAT= specifier and
6 the previous *image* to have entered the construct failed while executing it, the *stat-variable* becomes defined
7 with the value **STAT_FAILED_IMAGE** and execution of the construct continues normally. If any other error
8 condition occurs during execution of a **CRITICAL statement** that has a STAT= specifier, the *stat-variable*
9 becomes defined with a processor-dependent positive value other than **STAT_FAILED_IMAGE**.
- 10 11 If an error condition occurs during execution of an *image control statement* that does not contain the STAT=
11 specifier in a *sync-stat*, error termination is initiated.
- 12 12 If an ERRMSG= specifier appears in an *image control statement* and an error condition occurs, *errmsg-*
13 *variable* is assigned an explanatory message, as if by *intrinsic assignment*. If no such condition occurs, the
14 definition status and value of *errmsg-variable* are unchanged.
- 15 13 The set of error conditions that can occur in an *image control statement* is processor dependent.

NOTE

A processor might detect communication failure between images and treat it as an error condition. A processor might also treat an invalid set of images in a SYNC IMAGES statement as an error condition.

12 Input/output statements

12.1 Input/output concepts

- 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** (12.3.5) or a sequence of records, which provide an extra level of organization to the file. A file composed of records is called a **record file**. A file composed of **file storage units** is called a **stream file**. A processor may allow a file to be viewed both as a **record file** and as a **stream file**; in this case the relationship between the **file storage units** when viewed as a **stream file** and the records when viewed as a **record file** is processor dependent.
- 5 A file is either an **external file** (12.3) or an **internal file** (12.4).

12.2 Records

12.2.1 Definition of a record

- 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

What is called a “record” in Fortran is commonly called a “logical record”. There is no concept in Fortran of a “physical record.”

12.2.2 Formatted record

- 1 A formatted record consists of a sequence of characters that are representable in the processor; however, a processor may prohibit some control characters (6.1.1) from appearing in a formatted record. The length of a formatted record is measured in characters and depends primarily on the number of characters put into the record when it is written; however, it may depend on the processor and the external medium. The length may be zero. Formatted records shall be read or written only by **formatted input/output statements**.

12.2.3 Unformatted record

- 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](#) (12.3.5) and depends on the output list (12.6.3) used when it is written, as well as on the processor and the external medium. The length may be zero. Unformatted records shall be read or written only by [unformatted input/output statements](#).

12.2.4 Endfile record

- 1 An endfile record is written explicitly by the [ENDFILE statement](#); the file shall be [connected](#) for sequential access. An endfile record is written implicitly to a file [connected](#) for sequential access when the most recent [data transfer statement](#) referring to the file is an [output statement](#), no intervening [file positioning statement](#) referring to the file has been executed, and

- a [REWIND](#) or [BACKSPACE](#) statement references the [unit](#) to which the file is [connected](#), or
- the [unit](#) is closed, either explicitly by a [CLOSE statement](#), implicitly by [normal termination](#), or implicitly by another [OPEN statement](#) for the same [unit](#).

- 2 An endfile record shall occur only as the last record of a file. An endfile record does not have a length property.

NOTE

An endfile record does not necessarily have any physical embodiment. The processor can use a record count or any other means to register the position of the file at the time an [ENDFILE statement](#) is executed, so that it can take appropriate action when that position is reached again during a read operation. The endfile record, however it is implemented, is considered to exist for the [BACKSPACE statement](#) (12.8.2).

12.3 External files

12.3.1 External file concepts

- 1 An [external file](#) is any file that exists in a medium external to the program.
- 2 At any given time, there is a processor-dependent set of allowed access methods, a processor-dependent set of allowed forms, a processor-dependent set of allowed actions, and a processor-dependent set of allowed record lengths for a file.

NOTE 1

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 2

If different files are needed on each [image](#), using a different file name on each [image](#) will improve portability of the code. One technique is to incorporate the [image index](#) as part of the name.

- 4 An [external file](#) that is [connected](#) to a [unit](#) has a position property (12.3.4).

NOTE 3

For more explanatory information on external files , see C.8.1 .

12.3.2 File existence

- 1 At any given time, there is a processor-dependent set of [external files](#) that exist for a program. A file may be known to the processor, yet not exist for a program at a particular time.
- 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](#).

12.3.3 File access**12.3.3.1 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

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](#) ([12.5.4](#)) or when the file is created if the file is [preconnected](#) ([12.5.5](#)).

12.3.3.2 Sequential access

- 1 Sequential access is a method of accessing the records of an external [record file](#) in order.
- 2 While [connected](#) for sequential access, an [external file](#) has the following properties.
 - The order of the records is the order in which they were written if the direct access method is not a member of the set of allowed access methods for the file. If the direct access method is also a member of the set of allowed access methods for the file, the order of the records is the same as that specified for direct access. In this case, the first record accessible by sequential access is the record whose record number is 1 for direct access. The second record accessible by sequential access is the record whose record number is 2 for direct access, etc. A record that has not been written since the file was created shall not be read.
 - The records of the file are either all formatted or all unformatted, except that the last record of the file can be an endfile record. Unless the previous reference to the file was an [output statement](#), the last record, if any, of the file shall be an endfile record.
 - The records of the file shall be read or written only by sequential access [data transfer statements](#).

12.3.3.3 Direct access

- 1 Direct access is a method of accessing the records of an external [record file](#) in arbitrary order.

- 1 2 While **connected** for direct access, an **external file** has the following properties.
- 2 • Each record of the file is uniquely identified by a positive integer called the record number. The record
 - 3 number of a record is specified when the record is written. Once established, the record number of
 - 4 a record can never be changed. The order of the records is the order of their record numbers.
 - 5 • The records of the file are either all formatted or all unformatted. If the sequential access method
 - 6 is also a member of the set of allowed access methods for the file, its endfile record, if any, is not
 - 7 considered to be part of the file while it is **connected** for direct access. If the sequential access method
 - 8 is not a member of the set of allowed access methods for the file, the file shall not contain an endfile
 - 9 record.
 - 10 • The records of the file shall be read or written only by direct access **data transfer statements**.
 - 11 • All records of the file have the same length.
 - 12 • Records need not be read or written in the order of their record numbers. Any record may be written
 - 13 into the file while it is **connected** to a **unit**. For example, it is permissible to write record 3, even though
 - 14 records 1 and 2 have not been written. Any record may be read from the file while it is **connected** to
 - 15 a **unit**, provided that the record has been written since the file was created, and if a **READ statement**
 - 16 for this connection is permitted.
 - 17 • The records of the file shall not be read or written using list-directed formatting (13.10), namelist
 - 18 formatting (13.11), or a nonadvancing **data transfer statement** (12.3.4.2).

NOTE

A record cannot be deleted; however, a record can be rewritten.

19 12.3.3.4 Stream access

- 20 1 Stream access is a method of accessing the **file storage units** (12.3.5) of an external **stream file**.
- 21 2 The properties of an **external file connected** for stream access depend on whether the connection is for
- 22 unformatted or formatted access. While connected for stream access, the **file storage units** of the file shall
- 23 be read or written only by stream access **data transfer statements**.
- 24 3 While **connected** for unformatted stream access, an **external file** has the following properties.
- 25 • Each **file storage unit** in the file is uniquely identified by a positive integer called the position. The
 - 26 first **file storage unit** in the file is at position 1. The position of each subsequent **file storage unit** is
 - 27 one greater than that of its preceding **file storage unit**.
 - 28 • If it is possible to position the file, the **file storage units** need not be read or written in order of their
 - 29 position. For example, it might be permissible to write the **file storage unit** at position 3, even though
 - 30 the **file storage units** at positions 1 and 2 have not been written. Any **file storage unit** may be read
 - 31 from the file while it is **connected** to a **unit**, provided that the **file storage unit** has been written since
 - 32 the file was created, and if a **READ statement** for this connection is permitted.
- 33 4 While **connected** for formatted stream access, an **external file** has the following properties.
- 34 • Some **file storage units** of the file can contain record markers; this imposes a record structure on the
 - 35 file in addition to its stream structure. There might or might not be a record marker at the end of the
 - 36 file. If there is no record marker at the end of the file, the final record is incomplete.
 - 37 • No maximum length (12.5.6.16) is applicable to these records.
 - 38 • Writing an empty record with no record marker has no effect.

- Each **file storage unit** in the file is uniquely identified by a positive integer called the position. The first **file storage unit** in the file is at position 1. The relationship between positions of successive **file storage units** is processor dependent; not all positive integers need correspond to valid positions.
- If it is possible to position the file, the file position can be set to a position that was previously identified by the **POS= specifier** in an **INQUIRE statement**.
- A processor may prohibit some control characters (6.1.1) from appearing in a formatted **stream file**.

NOTE 1

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 2

There might be some character positions in the file that do not correspond to characters written; this is because on some processors a record marker could be written to the file as a carriage-return/line-feed or other sequence. The means of determining the position in a file **connected** for stream access is via the **POS= specifier** in an **INQUIRE statement** (12.10.2.23).

12.3.4 File position**12.3.4.1 General**

- 1 Execution of certain input/output statements affects the position of an **external file**. Certain circumstances can cause the position of a file to become indeterminate.
- 2 The initial point of a file is the position just before the first record or **file storage unit**. The terminal point is the position just after the last record or **file storage unit**. If there are no records or **file storage units** in the file, the initial point and the terminal point are the same position.
- 3 If a **record file** is positioned within a record, that record is the current record; otherwise, there is no current record.
- 4 Let n be the number of records in the file. If $1 < i \leq n$ and a file is positioned within the i th record or between the $(i - 1)$ th record and the i th record, the $(i - 1)$ th record is the preceding record. If $n \geq 1$ and the file is positioned at its terminal point, the preceding record is the n th and last record. If $n = 0$ or if a file is positioned at its initial point or within the first record, there is no preceding record.
- 5 If $1 \leq i < n$ and a file is positioned within the i th record or between the i th and $(i + 1)$ th record, the $(i + 1)$ th record is the next record. If $n \geq 1$ and the file is positioned at its initial point, the first record is the next record. If $n = 0$ or if a file is positioned at its terminal point or within the n th (last) record, there is no next record.
- 6 For a file **connected** for stream access, the file position is either between two **file storage units**, at the initial point of the file, at the terminal point of the file, or undefined.

12.3.4.2 Advancing and nonadvancing input/output

- 1 An advancing input/output statement always positions a **record file** after the last record read or written, unless there is an error condition.
- 2 A nonadvancing input/output statement may position a **record file** at a character position within the current record, or a subsequent record (13.8.2). Using nonadvancing input/output, it is possible to read or

write a record of the file by a sequence of [data transfer statements](#), each accessing a portion of the record. If a nonadvancing [output statement](#) leaves a file positioned within a current record and no further [output statement](#) is executed for the file before it is closed or a [BACKSPACE](#), [ENDFILE](#), or [REWIND](#) statement is executed for it, the effect is as if the [output statement](#) were the corresponding advancing [output statement](#).

12.3.4.3 File position prior to data transfer

- 1 The positioning of the file prior to data transfer depends on the method of access: sequential, direct, or stream.
- 2 For sequential access on input, if there is a current record, the file position is not changed. Otherwise, the file is positioned at the beginning of the next record and this record becomes the current record. Input shall not occur if there is no next record or if there is a current record and the last [data transfer statement](#) accessing the file performed output.
- 3 If the file contains an endfile record, the file shall not be positioned after the endfile record prior to data transfer. However, a [REWIND](#) or [BACKSPACE](#) statement may be used to reposition the file.
- 4 For sequential access on output, if there is a current record, the file position is not changed and the current record becomes the last record of the file. Otherwise, a new record is created as the next record of the file; this new record becomes the last and current record of the file and the file is positioned at the beginning of this record.
- 5 For direct access, the file is positioned at the beginning of the record specified by the [REC= specifier](#). This record becomes the current record.
- 6 For stream access, the file is positioned immediately before the [file storage unit](#) specified by the [POS= specifier](#); if there is no [POS= specifier](#), the file position is not changed.
- 7 File positioning for child [data transfer statements](#) is described in [12.6.4.8](#).

12.3.4.4 File position after data transfer

- 1 If an error condition ([12.11](#)) occurred, the position of the file is indeterminate. If no error condition occurred, but an end-of-file condition ([12.11](#)) occurred as a result of reading an endfile record, the file is positioned after the endfile record.
- 2 For unformatted stream input/output, if no error condition occurred, the file position is not changed. For unformatted stream output, if the file position exceeds the previous terminal point of the file, the terminal point is set to the file position.

NOTE 1

An unformatted stream [output statement](#) with a [POS= specifier](#) and an empty output list can have the effect of extending the terminal point of a file without actually writing any data.

- 3 For formatted stream input, if an end-of-file condition occurred, the file position is not changed.
- 4 For nonadvancing input, if no error condition or end-of-file condition occurred, but an end-of-record condition ([12.11](#)) occurred, the file is positioned after the record just read. If no error condition, end-of-file condition, or end-of-record condition occurred in a nonadvancing [input statement](#), the file position is not changed. If no error condition occurred in a nonadvancing [output statement](#), the file position is not changed.

- 1 5 In all other cases, the file is positioned after the record just read or written and that record becomes the
2 preceding record.
- 3 6 For a formatted stream [output statement](#), if no error condition occurred, the terminal point of the file is set
4 to the next position after the highest-numbered position to which a datum was transferred by the state-
5 ment.

NOTE 2

The highest-numbered position might not be the current one if the output involved a T, TL, TR, or X edit descriptor ([13.8.1](#)) and the statement is a nonadvancing [output statement](#).

6 12.3.5 File storage units

- 7 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
8 file position for stream access, the unit of record length for unformatted files, and the unit of file size for all
9 [external files](#).
- 10 2 Every value in a [stream file](#) or an unformatted [record file](#) shall occupy an integer number of [file storage](#)
11 [units](#); if the [stream](#) or [record](#) file is unformatted, this number shall be the same for all scalar values of the
12 same type and type parameters. The number of [file storage units](#) required for an item of a given type and
13 type parameters can be determined using the [IOLENGTH= specifier](#) of the [INQUIRE statement](#) ([12.10.3](#)).
- 14 3 For a file [connected](#) for unformatted stream access, the processor shall not have alignment restrictions that
15 prevent a value of any type from being stored at any positive integer file position.
- 16 4 The number of bits in a [file storage unit](#) is given by the constant FILE_STORAGE_SIZE ([16.10.2.11](#)) defined
17 in the intrinsic module [ISO_FORTRAN_ENV](#). It is recommended that the [file storage unit](#) be an 8-bit octet
18 where this choice is practical.

NOTE

The requirement that every data value occupy an integer number of [file storage units](#) implies that data items inherently smaller than a [file storage unit](#) will require padding. This suggests that the [file storage unit](#) be small to avoid wasted space. Ideally, the file storage unit would be chosen such that padding is never required. A [file storage unit](#) of one bit would always meet this goal, but would likely be impractical because of the alignment requirements.

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.

19 12.4 Internal files

- 20 1 [Internal files](#) provide a means of transferring and converting data from internal storage to internal storage.
- 21 2 An [internal file](#) is a [record file](#) with the following properties.
- 22 • The file is a variable of default, [ASCII](#), or [ISO 10646](#) character kind that is not an [array section](#) with a
23 [vector subscript](#).
 - 24 • A record of an [internal file](#) is a scalar character variable.
 - 25 • If the file is a scalar character variable, it consists of a single record whose length is the same as
26 the length of the scalar character variable. If the file is a character array, it is treated as a sequence
27 of character array elements. Each array element, if any, is a record of the file. The ordering of the

records of the file is the same as the ordering of the array elements in the array (9.5.3.3) or the [array section](#) (9.5.3.4). 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 internal file is an allocatable, deferred-length character scalar variable, it is assigned the characters written by [intrinsic assignment](#), allocating or reallocating to have length equal to the number of characters written if necessary.
 - Otherwise, 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 shall be read only if the record is defined.
- A record of an [internal file](#) can become defined (or undefined) by means other than an [output statement](#). For example, the character variable can become defined by a character [assignment statement](#).
- An [internal file](#) is always positioned at the beginning of the first record prior to data transfer, except for child [data transfer statements](#) (12.6.4.8). This record becomes the current record.
- The initial value of a connection mode (12.5.2) is the value that would be implied by an initial [OPEN statement](#) without the corresponding keyword.
- Reading and writing records shall be accomplished only by sequential access formatted [data transfer statements](#).
- An [internal file](#) shall not be specified as the [unit](#) in a [CLOSE](#), [INQUIRE](#), or [OPEN](#) statement.

12.5 File connection

12.5.1 Referring to a file

- 1 A [unit](#), specified by an [io-unit](#), provides a means for referring to a file.

R1201 *io-unit* **is** *file-unit-number*
 or ***
 or *internal-file-variable*

R1202 *file-unit-number* **is** *scalar-int-expr*

R1203 *internal-file-variable* **is** *char-variable*

C1201 (R1203) The *char-variable* shall not be an [array section](#) with a [vector subscript](#).

C1202 (R1203) The *char-variable* shall be default character, [ASCII character](#), or [ISO 10646 character](#).

- 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, the `unit` argument of an active [defined input/output](#) procedure (12.6.4.8), a [NEWUNIT](#) value (12.5.6.13), or equal to one of the [named constants](#) `INPUT_UNIT`, `OUTPUT_UNIT`, or `ERROR_UNIT` of the intrinsic module [ISO_FORTRAN_ENV](#) (16.10.2). 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 On an [image](#), the [external unit](#) identified by a particular value of a *scalar-int-expr* is the same [external unit](#) in all [program units](#).

NOTE 1

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** in the **initial team** only (12.6.4.3); it is not **preconnected** on any other **image**. This **unit** is also identified by the value of the **named constant INPUT_UNIT** of the intrinsic module **ISO_FORTRAN_ENV** (16.10.2.13). This **unit** is also used by a **READ statement** without an **io-control-spec-list**. In a **WRITE statement**, an **io-unit** that is an asterisk identifies an **external unit** that is **preconnected** for sequential formatted output. This **unit** is also identified by the value of the **named constant OUTPUT_UNIT** of the intrinsic module **ISO_FORTRAN_ENV** (16.10.2.24). This **unit** is also used by a **PRINT statement**.
- 5 This document identifies a processor-dependent **external unit** for the purpose of error reporting. This **unit** shall be **preconnected** for sequential formatted output. The processor may define this to be the same as the output **unit** identified by an asterisk. This **unit** is also identified by a unit number defined by the **named constant ERROR_UNIT** of the intrinsic module **ISO_FORTRAN_ENV**.

NOTE 2

Even though **OUTPUT_UNIT** is connected to a separate file on each **image**, it is expected that the processor could merge the sequences of records from these files into a single sequence of records that is sent to the physical device associated with this **unit**, such as the user's terminal. If **ERROR_UNIT** is associated with the same physical device, the sequences of records from files connected to **ERROR_UNIT** on each of the **images** could be merged into the same sequence generated from the **OUTPUT_UNIT** files. Otherwise, it is expected that the sequence of records in the files connected to **ERROR_UNIT** on each **image** could be merged into a single sequence of records that is sent to the physical device associated with **ERROR_UNIT**.

12.5.2 Connection modes

- 1 A connection for formatted input/output has several changeable modes: these are the blank interpretation mode (13.8.7), delimiter mode (13.10.4, 13.11.4.2), sign mode (13.8.4), leading zero mode (13.8.5), decimal edit mode (13.8.9), input/output rounding mode (13.7.2.3.8), pad mode (12.6.4.5.3), and scale factor (13.8.6). 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** can be changed by a subsequent **OPEN statement** that modifies the connection.
- 5 The modes of a connection can 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 im-

diately before the [data transfer statement](#) was executed.

12.5.3 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.

12.5.4 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 ([12.3.2](#)).

NOTE 1

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 document defines means of portable interoperability with C. C streams are described in ISO/IEC 9899:2018, 7.21.2. 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 If input/output operations are performed on more than one unit while they are connected to the same [external file](#), the results are processor dependent.

7 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 2

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.

- 1 8 An **internal unit** is always **connected** to the **internal file** designated by the variable that identifies the **unit**.

NOTE 3

For more explanatory information on file connection properties, see C.8.4.

12.5.5 Preconnection

- 1 Preconnection means that the **unit** is **connected** to a file at the beginning of execution of the program and
 4 therefore it may be specified in input/output statements without the prior execution of an **OPEN statement**.

12.5.6 OPEN statement

12.5.6.1 General

- 1 An OPEN statement initiates or modifies the connection between an **external file** and a specified **unit**. The
 8 OPEN statement can be used to connect an existing file to a **unit**, create a file that is **preconnected**, create a
 9 file and connect it to a **unit**, or change certain modes of a connection between a file and a **unit**.
- 2 An **external unit** may be **connected** by an OPEN statement in the main program or any subprogram.
- 3 If the file to be **connected** to the **unit** does not exist but is the same as the file to which the **unit** is **precon-**
 12 **connected**, 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
 14 as if a **CLOSE statement** without a **STATUS= specifier** had been executed for the **unit** immediately prior to
 15 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
 17 **FILE= specifier** is not included in such an OPEN statement, the file to be **connected** to the **unit** is the same
 18 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
 20 is not established and values for any changeable modes (12.5.2) specified come into effect for the estab-
 21 lished connection; the current file position is unaffected. Before any effect on changeable modes, a wait
 22 operation is performed for any pending asynchronous data transfer operations for the specified **unit**. If the
 23 **POSITION= specifier** appears in such an OPEN statement, the value specified shall not disagree with the
 24 current position of the file. If the **STATUS= specifier** is included in such an OPEN statement, it shall be spe-
 25 cified with the value OLD. Other than **ERR=**, **IOSTAT=**, and **IOMSG=**, and the changeable modes, the values
 26 of all other specifiers in such an OPEN statement shall not differ from those in effect for the established
 27 connection.
- 7 A **STATUS= specifier** with a value of OLD is always allowed when the file to be **connected** to the **unit** is the
 29 same as the file to which the **unit** is **connected**. In this case, if the status of the file was SCRATCH before
 30 execution of the OPEN statement, the file will still be deleted when the **unit** is closed, and the file is still
 31 considered to have a status of SCRATCH.

12.5.6.2 Syntax of the OPEN statement

- 33 R1204 *open-stmt* **is** OPEN (*connect-spec-list*)
- 34 R1205 *connect-spec* **is** [UNIT =] *file-unit-number*
- 35 **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 = *stat-variable*
or LEADING_ZERO = *scalar-default-char-expr*
or NEWUNIT = *scalar-int-variable*
or PAD = *scalar-default-char-expr*
or POSITION = *scalar-default-char-expr*
or RECL = *scalar-int-expr*
or ROUND = *scalar-default-char-expr*
or SIGN = *scalar-default-char-expr*
or STATUS = *scalar-default-char-expr*

R1206 *file-name-expr* **is** *scalar-default-char-expr*

R1207 *iomsg-variable* **is** *scalar-default-char-variable*

C1203 No specifier shall appear more than once in a given *connect-spec-list*.

C1204 (R1204) If the **NEWUNIT= specifier** does not appear, a *file-unit-number* shall be specified; if the optional characters UNIT= are omitted, the *file-unit-number* shall be the first item in the *connect-spec-list*.

C1205 (R1204) If a **NEWUNIT= specifier** appears, a *file-unit-number* shall not appear.

C1206 (R1204) The *label* used in the **ERR= specifier** shall be the statement label of a **branch target statement** that appears in the same **inclusive scope** as the OPEN statement.

1 Some specifiers that require a *scalar-default-char-expr* 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.

2 The **IOSTAT=**, **ERR=**, and **IOMSG=** specifiers are described in 12.11.

NOTE 1

An example of an OPEN statement is:

```
OPEN (10, FILE = 'employee.names', ACTION = 'READ', PAD = 'YES')
```

NOTE 2

For more explanatory information on the OPEN statement, see C.8.3.

12.5.6.3 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.

12.5.6.4 ACTION= specifier in the OPEN statement

The *scalar-default-char-expr* shall evaluate to READ, WRITE, or READWRITE. READ specifies that the WRITE, PRINT, and ENDFILE statements shall not refer to this connection. WRITE specifies that READ statements shall not refer to this connection. READWRITE permits any input/output statement to refer to this connection. If this specifier is omitted, the default value is processor dependent. If READWRITE is included in the set of allowable actions for a file, both READ and WRITE also shall be included in the set of allowed actions for that file. For an existing file, the specified action shall be included in the set of allowed actions for the file. For a new file, the processor creates the file with a set of allowed actions that includes the specified action.

12.5.6.5 ASYNCHRONOUS= specifier in the OPEN statement

The *scalar-default-char-expr* shall evaluate to YES or NO. If YES is specified, asynchronous input/output on the *unit* is allowed. If NO is specified, asynchronous input/output on the *unit* is not allowed. If this specifier is omitted, the default value is NO.

12.5.6.6 BLANK= specifier in the OPEN statement

The *scalar-default-char-expr* shall evaluate to NULL or ZERO. The BLANK= specifier is permitted only for a connection for formatted input/output. It specifies the blank interpretation mode (13.8.7, 12.6.2.6) for input for this connection. This mode has no effect on output. It is a changeable mode (12.5.2). If this specifier is omitted in an OPEN statement that initiates a connection, the default value is NULL.

12.5.6.7 DECIMAL= specifier in the OPEN statement

The *scalar-default-char-expr* shall evaluate to COMMA or POINT. The DECIMAL= specifier is permitted only for a connection for formatted input/output. It specifies the decimal edit mode (13.6, 13.8.9, 12.6.2.7) for this connection. It is a changeable mode (12.5.2). If this specifier is omitted in an OPEN statement that initiates a connection, the default value is POINT.

12.5.6.8 DELIM= specifier in the OPEN statement

The *scalar-default-char-expr* shall evaluate to APOSTROPHE, QUOTE, or NONE. The DELIM= specifier is permitted only for a connection for formatted input/output. It specifies the delimiter mode (12.6.2.8) for list-directed (13.10.4) and namelist (13.11.4.2) output for the connection. This mode has no effect on input. It is a changeable mode (12.5.2). If this specifier is omitted in an OPEN statement that initiates a connection, the default value is NONE.

12.5.6.9 ENCODING= specifier in the OPEN statement

The *scalar-default-char-expr* shall evaluate to UTF-8 or DEFAULT. The ENCODING= specifier is permitted only for a connection for formatted input/output. 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.

12.5.6.10 FILE= specifier in the OPEN statement

1 The value of the FILE= specifier is the name of the file to be **connected** to the specified **unit**. Any trailing blanks are ignored. The *file-name-expr* shall be a name that is allowed by the processor. The interpretation of case is processor dependent.

2 This specifier shall appear if the **STATUS= specifier** has the value NEW or REPLACE. This specifier shall not appear if the **STATUS= specifier** has the value SCRATCH. If the **STATUS= specifier** has the value OLD, this specifier shall appear unless the **unit** is **connected** and the file **connected** to the **unit** exists. 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.

12.5.6.11 FORM= specifier in the OPEN statement

1 The *scalar-default-char-expr* shall evaluate to FORMATTED or UNFORMATTED. The FORM= specifier determines whether the file is being connected for formatted or unformatted input/output. If this specifier is omitted, the default value is UNFORMATTED if the file is being connected for direct access or stream access, and the default value is FORMATTED if the file is being connected for sequential access. For an existing file, the specified form shall be included in the set of allowed forms for the file. For a new file, the processor creates the file with a set of allowed forms that includes the specified form.

12.5.6.12 LEADING_ZERO= specifier in the OPEN statement

1 The *scalar-default-char-expr* shall evaluate to one of PRINT, SUPPRESS, or PROCESSOR_DEFINED. The LEADING_ZERO= specifier is permitted only for a connection for formatted input/output. It specifies the leading zero mode (13.8.5, 12.6.2.10) for this connection. It is a changeable mode (12.5.2). If this specifier is omitted in an OPEN statement that initiates a connection, the default value is PROCESSOR_DEFINED.

12.5.6.13 NEWUNIT= specifier in the OPEN statement

1 If this specifier appears in an OPEN statement, either the **FILE= specifier** shall appear, or the **STATUS= specifier** shall appear with a value of SCRATCH.

2 The variable is defined with a processor determined NEWUNIT value if no error condition occurs during the execution of the OPEN statement. If an error condition occurs, the processor shall not change the value of the variable.

3 A NEWUNIT value is a negative number, and shall not be equal to -1 , any of the **named constants** **ERROR_UNIT**, **INPUT_UNIT**, or **OUTPUT_UNIT** from the intrinsic module **ISO_FORTRAN_ENV** (16.10.2), any value used by the processor for the **unit** argument to a **defined input/output** procedure, nor any previous NEWUNIT value that identifies a file that is **connected**. The **unit** identified by a NEWUNIT value shall not be **preconnected**.

12.5.6.14 PAD= specifier in the OPEN statement

1 The *scalar-default-char-expr* shall evaluate to YES or NO. The PAD= specifier is permitted only for a connection for formatted input/output. It specifies the pad mode (12.6.4.5.3, 12.6.2.11) for input for this connection. This mode has no effect on output. It is a changeable mode (12.5.2). If this specifier is omitted in an OPEN statement that initiates a connection, the default value is YES.

12.5.6.15 POSITION= specifier in the OPEN statement

1 The *scalar-default-char-expr* shall evaluate to ASIS, REWIND, or APPEND. The connection shall be for sequential or stream access. A new file is positioned at its initial point. REWIND positions an existing file at its initial point. APPEND positions an existing file such that the endfile record is the next record, if it has one. If an existing file does not have an endfile record, APPEND positions the file at its terminal point. ASIS leaves the position unchanged if the file exists and already is [connected](#). If the file exists but is not connected, the position resulting from ASIS is processor dependent. If this specifier is omitted, the default value is ASIS.

12.5.6.16 RECL= specifier in the OPEN statement

1 The value of the RECL= specifier shall be positive. It specifies the length of each record in a file being connected for direct access, or specifies the maximum length of a record in a file being connected for sequential access. This specifier shall not appear when a file is being connected for stream access. This specifier shall appear when a file is being connected for direct access. If this specifier is omitted when a file is being connected for sequential access, the default value is processor dependent. If the file is being connected for formatted input/output, the length is the number of characters for all records that contain only characters of default kind. When a record contains any nondefault characters, the effect of the RECL= specifier is processor dependent. If the file is being connected for unformatted input/output, the length is measured in [file storage units](#). For an existing file, the value of the RECL= specifier shall be included in the set of allowed record lengths for the file. For a new file, the processor creates the file with a set of allowed record lengths that includes the specified value.

12.5.6.17 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 ([13.7.2.3.8](#), [12.6.2.14](#)) for this connection. It is a changeable mode ([12.5.2](#)). If this specifier is omitted in an OPEN statement that initiates a connection, the input/output rounding mode is processor dependent; it shall be one of the above modes.

NOTE

A processor is free to select any input/output rounding mode for the default mode. The mode might correspond to UP, DOWN, ZERO, NEAREST, or COMPATIBLE; or it might be a completely different input/output rounding mode.

12.5.6.18 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 ([13.8.4](#), [12.6.2.15](#)) for this connection. It is a changeable mode ([12.5.2](#)). If this specifier is omitted in an OPEN statement that initiates a connection, the default value is PROCESSOR_DEFINED.

12.5.6.19 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

1 name, and the status is changed to OLD. If SCRATCH is specified, the file is created and connected to the
 2 specified **unit** for use by the program but is deleted at the execution of a **CLOSE statement** referring to the
 3 same **unit** or at the **normal termination** of the program.

4 3 If UNKNOWN is specified, the status is processor dependent. If this specifier is omitted, the default value
 5 is UNKNOWN.

NOTE

SCRATCH cannot be specified if the **FILE= specifier** appears (12.5.6.10).

6 12.5.7 CLOSE statement

7 12.5.7.1 General

- 8 1 The CLOSE statement is used to terminate the connection of a specified **unit** to an **external file**.
- 9 2 Execution of a CLOSE statement for a **unit** may occur in any **program unit** of a program and need not occur
 10 in the same **program unit** as the execution of an **OPEN statement** referring to that **unit**.
- 11 3 Execution of a CLOSE statement performs a wait operation for any pending asynchronous data transfer
 12 operations for the specified **unit**.
- 13 4 Execution of a CLOSE statement specifying a **unit** that does not exist, exists but is connected to a file that
 14 does not exist, or has no file **connected** to it, is permitted and affects no file or **unit**.
- 15 5 After a **unit** has been disconnected by execution of a CLOSE statement, it may be **connected** again within
 16 the same program, either to the same file or to a different file. After a named file has been disconnected by
 17 execution of a CLOSE statement, it may be **connected** again within the same program, either to the same
 18 **unit** or to a different **unit**, provided that the file still exists.
- 19 6 During the completion step (5.3.7) of **normal termination**, all **units** that are **connected** are closed. Each **unit**
 20 is closed with status KEEP unless the file status prior to **termination of execution** was SCRATCH, in which
 21 case the **unit** is closed with status DELETE.

NOTE

The effect is as though a CLOSE statement without a STATUS= specifier were executed on each **connected unit**.

22 12.5.7.2 Syntax

23	R1208 <i>close-stmt</i>	is CLOSE (<i>close-spec-list</i>)
24	R1209 <i>close-spec</i>	is [UNIT =] <i>file-unit-number</i>
25		or IOSTAT = <i>stat-variable</i>
26		or IOMSG = <i>iomsg-variable</i>
27		or ERR = <i>label</i>
28		or STATUS = <i>scalar-default-char-expr</i>

29 C1207 No specifier shall appear more than once in a given *close-spec-list*.

30 C1208 A *file-unit-number* shall be specified in a *close-spec-list*; if the optional characters UNIT= are omit-
 31 ted, the *file-unit-number* shall be the first item in the *close-spec-list*.

1 C1209 (R1209) The *label* used in the *ERR= specifier* shall be the statement label of a *branch target state-*
 2 *ment* that appears in the same *inclusive scope* as the CLOSE statement.

3 1 The *scalar-default-char-expr* has a limited list of character values. Any trailing blanks are ignored. The
 4 value specified is without regard to case.

5 2 The *IOSTAT=*, *ERR=*, and *IOMSG=* specifiers are described in 12.11.

NOTE

An example of a CLOSE statement is:

```
CLOSE (10, STATUS = 'KEEP')
```

6 12.5.7.3 STATUS= specifier in the CLOSE statement

7 1 The *scalar-default-char-expr* shall evaluate to KEEP or DELETE. The STATUS= specifier determines the dis-
 8 position of the file that is *connected* to the specified *unit*. KEEP shall not be specified for a file whose status
 9 prior to execution of a CLOSE statement is SCRATCH. If KEEP is specified for a file that exists, the file con-
 10 tinues to exist after the execution of a CLOSE statement. If KEEP is specified for a file that does not exist,
 11 the file will not exist after the execution of a CLOSE statement. If DELETE is specified, the file will not exist
 12 after the execution of a CLOSE statement. If this specifier is omitted, the default value is KEEP, unless the
 13 file status prior to execution of the CLOSE statement is SCRATCH, in which case the default value is DELETE.
 14

15 12.6 Data transfer statements

16 12.6.1 Form of input and output statements

17 1 The READ statement is the data transfer input statement. The WRITE statement and the PRINT statement
 18 are the data transfer output statements.

19 R1210 *read-stmt* **is** READ (*io-control-spec-list*) [*input-item-list*]

20 **or** READ *format* [, *input-item-list*]

21 R1211 *write-stmt* **is** WRITE (*io-control-spec-list*) [*output-item-list*]

22 R1212 *print-stmt* **is** PRINT *format* [, *output-item-list*]

NOTE 1

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)
```

NOTE 2

A statement of the form

```
READ (name)
```

where *name* is the name of a default character variable is a formatted input statement. The format expression “(*name*)” is the *format*. The statement cannot be an input statement that specifies an internal file because of C1221.

12.6.2 Control information list

12.6.2.1 Syntax

1 A control information list is an *io-control-spec-list*. It governs data transfer.

```

R1213 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 = stat-variable
                                or  LEADING_ZERO = scalar-default-char-expr
                                or  PAD = scalar-default-char-expr
                                or  POS = scalar-int-expr
                                or  REC = scalar-int-expr
                                or  ROUND = scalar-default-char-expr
                                or  SIGN = scalar-default-char-expr
                                or  SIZE = scalar-int-variable

```

R1214 *id-variable* is *scalar-int-variable*

C1210 No specifier shall appear more than once in a given *io-control-spec-list*.

C1211 An *io-unit* shall be specified in an *io-control-spec-list*; if the optional characters UNIT= are omitted, the *io-unit* shall be the first item in the *io-control-spec-list*.

C1212 (R1213) A DELIM=, LEADING_ZERO=, or SIGN= specifier shall not appear in a *read-stmt*.

C1213 (R1213) A BLANK=, PAD=, END=, EOR=, or SIZE= specifier shall not appear in a *write-stmt*.

C1214 A SIZE= specifier shall not appear in a list-directed or namelist *input statement*.

C1215 (R1213) The *label* in the ERR=, EOR=, or END= specifier shall be the statement label of a *branch target statement* that appears in the same *inclusive scope* as the data transfer statement.

C1216 (R1213) A *namelist-group-name* shall be the name of a namelist group.

C1217 (R1213) A *namelist-group-name* shall not appear if a REC= specifier, *format*, *input-item-list*, or an *output-item-list* appears in the data transfer statement.

C1218 (R1213) 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*.

C1219 (R1213) If *namelist-group-name* appears without a preceding NML=, it shall be the second item in

- 1 the *io-control-spec-list* and the first item shall be *io-unit*.
- 2 C1220 (R1213) If *io-unit* is not a *file-unit-number*, the *io-control-spec-list* shall not contain a *REC= specifier*
3 or a *POS= specifier*.
- 4 C1221 (R1213) If *io-unit* is an *internal-file-variable*, the *io-control-spec-list* shall contain a *format* or a
5 *namelist-group-name*.
- 6 C1222 (R1213) If the *REC= specifier* appears, an *END= specifier* shall not appear, and the *format*, if any,
7 shall not be an asterisk.
- 8 C1223 (R1213) An *ADVANCE= specifier* shall appear only in a formatted sequential or stream data trans-
9 fer statement with explicit format specification (13.2) whose *io-control-spec-list* does not contain
10 an *internal-file-variable* as the *io-unit*.
- 11 C1224 (R1213) If an *EOR= specifier* appears, an *ADVANCE= specifier* also shall appear.
- 12 C1225 (R1213) The *scalar-default-char-constant-expr* in an *ASYNCHRONOUS= specifier* shall have the
13 value YES or NO.
- 14 C1226 (R1213) An *ASYNCHRONOUS= specifier* with a value YES shall not appear unless *io-unit* is a *file-*
15 *unit-number*.
- 16 C1227 (R1213) If an *ID= specifier* appears, an *ASYNCHRONOUS= specifier* with the value YES shall also
17 appear.
- 18 C1228 (R1213) If a *POS= specifier* appears, the *io-control-spec-list* shall not contain a *REC= specifier*.
- 19 C1229 (R1213) If a *DECIMAL=*, *BLANK=*, *LEADING_ZERO=*, *PAD=*, *SIGN=*, or *ROUND=* specifier appears, a
20 *format* or *namelist-group-name* shall also appear.
- 21 C1230 (R1213) If a *DELIM= specifier* appears, either *format* shall be an asterisk or *namelist-group-name*
22 shall appear.
- 23 C1231 (R1214) The *scalar-int-variable* shall have a decimal exponent range no smaller than that of default
24 integer.
- 25 2 If an *EOR= specifier* appears, an *ADVANCE= specifier* with the value NO shall also appear.
- 26 3 If the data transfer statement contains a *format* or *namelist-group-name*, the statement is a formatted in-
27 put/output statement; otherwise, it is an unformatted input/output statement.
- 28 4 The *ADVANCE=*, *ASYNCHRONOUS=*, *DECIMAL=*, *BLANK=*, *DELIM=*, *LEADING_ZERO=*, *PAD=*,
29 *ROUND=*, and *SIGN=* specifiers have a limited list of character values. Any trailing blanks are ignored. The
30 values specified are without regard to case.
- 31 5 The *IOSTAT=*, *ERR=*, *EOR=*, *END=*, and *IOMSG=* specifiers are described in 12.11.

NOTE

An example of a READ statement is:

```
READ (IOSTAT = IOS, UNIT = 6, FMT = '(10F8.2)') A, B
```

12.6.2.2 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.

R1215 *format* **is** *default-char-expr*
 or *label*
 or *

C1232 (R1215) The *label* shall be the label of a **FORMAT statement** that appears in the same **inclusive scope** as the statement containing the FMT= specifier.

- 2 The *default-char-expr* shall evaluate to a valid format specification (13.2.1 and 13.2.2).
 3 If *default-char-expr* is an array, it is treated as if all of the elements of the array were specified in array element order and were concatenated.
 4 If *format* is *, the statement is a list-directed input/output statement.

NOTE

An example in which the format is a character expression is:

```
READ (6, FMT = "(" // CHAR_FMT // ")") X, Y, Z
```

where CHAR_FMT is a default character variable.

12.6.2.3 NML= specifier in a data transfer statement

- 1 The NML= specifier supplies the *namelist-group-name* (8.9). This name identifies a particular collection of data objects on which transfer is to be performed.
 2 If a *namelist-group-name* appears, the statement is a namelist input/output statement.

12.6.2.4 ADVANCE= specifier in a data transfer statement

- 1 The *scalar-default-char-expr* shall evaluate to YES or NO. The ADVANCE= specifier determines whether advancing input/output occurs for a nonchild data transfer statement. If YES is specified for a nonchild data transfer statement, advancing input/output occurs. If NO is specified, nonadvancing input/output occurs (12.3.4.2). If this specifier is omitted from a nonchild data transfer statement that allows the specifier, the default value is YES. A formatted child data transfer statement is a nonadvancing input/output statement, and any ADVANCE= specifier is ignored.

12.6.2.5 ASYNCHRONOUS= specifier in a data transfer statement

- 1 The ASYNCHRONOUS= specifier determines whether this data transfer statement is synchronous or asynchronous. If YES is specified, the statement and the input/output operation are asynchronous. If NO is specified or if the specifier is omitted, the statement and the input/output operation are synchronous.
 2 Asynchronous input/output is permitted only for **external files** opened with an ASYNCHRONOUS= specifier with the value YES in the **OPEN statement**.

NOTE 1

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**.

NOTE 1 (cont.)

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.

- 1 3 The processor may perform an asynchronous data transfer operation asynchronously, but it is not required
 2 to do so. For each [external file](#), records and [file storage units](#) read or written by asynchronous data transfer
 3 statements are read, written, and processed in the same order as they would have been if the data trans-
 4 fer statements were synchronous. The documentation of the Fortran processor should describe when in-
 5 put/output will be performed asynchronously.
- 6 4 If a variable is used in an asynchronous data transfer statement as
- 7 • an item in an input/output list,
 - 8 • a group object in a namelist, or
 - 9 • a SIZE= specifier,
- 10 the [base object](#) of the *data-ref* is implicitly given the [ASYNCHRONOUS attribute](#) in the [scoping unit](#) of the
 11 data transfer statement. This attribute may be confirmed by explicit declaration.
- 12 5 When an asynchronous input/output statement is executed, the set of [storage units](#) specified by the item
 13 list or NML= specifier, plus the [storage units](#) specified by the SIZE= specifier, is defined to be the pending
 14 input/output storage sequence for the data transfer operation.

NOTE 2

A pending input/output storage sequence is not necessarily a [contiguous](#) set of [storage units](#).

- 15 6 A pending input/output storage sequence affector is a variable of which any part is associated with a [stor-](#)
 16 [age unit](#) in a pending input/output storage sequence.

12.6.2.6 BLANK= specifier in a data transfer statement

- 18 1 The *scalar-default-char-expr* shall evaluate to NULL or ZERO. The BLANK= specifier temporarily changes
 19 ([12.5.2](#)) the blank interpretation mode ([13.8.7](#), [12.5.6.6](#)) for the connection. If the specifier is omitted, the
 20 mode is not changed.

12.6.2.7 DECIMAL= specifier in a data transfer statement

- 22 1 The *scalar-default-char-expr* shall evaluate to COMMA or POINT. The DECIMAL= specifier temporarily chan-
 23 ges ([12.5.2](#)) the decimal edit mode ([13.6](#), [13.8.9](#), [12.5.6.7](#)) for the connection. If the specifier is omitted, the
 24 mode is not changed.

12.6.2.8 DELIM= specifier in a data transfer statement

- 26 1 The *scalar-default-char-expr* shall evaluate to APOSTROPHE, QUOTE, or NONE. The DELIM= specifier tem-
 27 porarily changes ([12.5.2](#)) the delimiter mode ([13.10.4](#), [13.11.4.2](#), [12.5.6.8](#)) for the connection. If the spe-
 28 cifier is omitted, the mode is not changed.

12.6.2.9 ID= specifier in a data transfer statement

- 30 1 Successful execution of an asynchronous data transfer statement containing an ID= specifier causes the
 31 variable specified in the ID= specifier to become defined with a processor determined value. If this value is
 32 zero, the data transfer operation has been completed. A nonzero value is referred to as the identifier of the

1 data transfer operation. This identifier is different from the identifier of any other pending data transfer
 2 operation for this [unit](#). It can be used in a subsequent [WAIT](#) or [INQUIRE](#) statement to identify the particular
 3 data transfer operation.

4 2 If an error condition occurs during the execution of a data transfer statement containing an ID= specifier,
 5 the variable specified in the ID= specifier becomes undefined.

6 3 A child data transfer statement shall not specify the ID= specifier.

7 **12.6.2.10 LEADING_ZERO= specifier in a data transfer statement**

8 1 The *scalar-default-char-expr* shall evaluate to PRINT, SUPPRESS, or PROCESSOR_DEFINED. The LEADING_
 9 ZERO= specifier temporarily changes ([12.5.2](#)) the leading zero mode ([13.8.5](#), [12.5.6.12](#)) for the connection.
 10 If the specifier is omitted, the mode is not changed.

11 **12.6.2.11 PAD= specifier in a data transfer statement**

12 1 The *scalar-default-char-expr* shall evaluate to YES or NO. The PAD= specifier temporarily changes ([12.5.2](#))
 13 the pad mode ([12.6.4.5.3](#), [12.5.6.14](#)) for the connection. If the specifier is omitted, the mode is not changed.

14 **12.6.2.12 POS= specifier in a data transfer statement**

15 1 The POS= specifier specifies the file position in [file storage units](#). This specifier shall not appear in a data
 16 transfer statement unless the statement specifies a [unit connected](#) for stream access. A child data transfer
 17 statement shall not specify this specifier.

18 2 A processor may prohibit the use of POS= with particular files that do not have the properties necessary
 19 to support random positioning. A processor may also prohibit positioning a particular file to any position
 20 prior to its current file position if the file does not have the properties necessary to support such position-
 21 ing.

NOTE

A unit that is connected to a device or data stream might not be positionable.

22 3 If the file is [connected](#) for formatted stream access, the file position specified by POS= shall be equal to either
 23 1 (the beginning of the file) or a value previously returned by a [POS= specifier](#) in an [INQUIRE statement](#) for
 24 the file.

25 **12.6.2.13 REC= specifier in a data transfer statement**

26 1 The REC= specifier specifies the number of the record that is to be read or written. This specifier shall ap-
 27 pear only in a data transfer statement that specifies a [unit connected](#) for direct access; it shall not appear in
 28 a child data transfer statement. If the *io-control-spec-list* contains a REC= specifier, the statement is a direct
 29 access data transfer statement. A child data transfer statement is a direct access data transfer statement
 30 if the parent is a direct access data transfer statement. Any other data transfer statement is a sequential
 31 access data transfer statement or a stream access data transfer statement, depending on whether the file
 32 connection is for sequential access or stream access.

33 **12.6.2.14 ROUND= specifier in a data transfer statement**

34 1 The *scalar-default-char-expr* shall evaluate to one of UP, DOWN, ZERO, NEAREST, COMPATIBLE or PRO-
 35 CESSOR_DEFINED. The ROUND= specifier temporarily changes ([12.5.2](#)) the input/output rounding mode

(13.7.2.3.8, 12.5.6.17) for the connection. If the specifier is omitted, the mode is not changed.

12.6.2.15 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 (12.5.2) the sign mode (13.8.4, 12.5.6.18) for the connection. If the specifier is omitted, the mode is not changed.

12.6.2.16 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.

12.6.3 Data transfer input/output list

1 An input/output list specifies the entities whose values are transferred by a data transfer statement.

R1216 *input-item* is *variable*
or *io-implied-do*

R1217 *output-item* is *expr*
or *io-implied-do*

R1218 *io-implied-do* is (*io-implied-do-object-list* , *io-implied-do-control*)

R1219 *io-implied-do-object* is *input-item*
or *output-item*

R1220 *io-implied-do-control* is *do-variable* = *scalar-int-expr* , ■
■ *scalar-int-expr* [, *scalar-int-expr*]

C1233 (R1216) A variable that is an *input-item* shall not be a whole *assumed-size* array.

C1234 (R1219) In an *input-item-list*, an *io-implied-do-object* shall be an *input-item*. In an *output-item-list*, an *io-implied-do-object* shall be an *output-item*.

C1235 (R1217) 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 1

A constant, an expression involving operators or function references that does not have a pointer result, or an expression enclosed in parentheses cannot 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 2

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).

- 1 4 If an input item or an output item is [allocatable](#), it shall be allocated.
- 2 5 A list item shall not be [polymorphic](#) unless it is processed by a [defined input/output](#) procedure (12.6.4.8).
- 3 6 A list item that is of an enumeration type shall not appear in a list-directed data transfer statement. In a
4 formatted data transfer statement, it shall correspond to an I, B, O, or Z edit descriptor.
- 5 7 The [do-variable](#) of an [io-implied-do](#) that is in another [io-implied-do](#) shall not appear as, nor be associated
6 with, the [do-variable](#) of the containing [io-implied-do](#).
- 7 8 The following rules describing whether to expand an input/output list item are re-applied to each expanded
8 list item until none of the rules apply.
 - 9 • If an array appears as an input/output list item, it is treated as if the elements, if any, were specified
10 in array element order (9.5.3.3). However, no element of that array shall affect the value of any ex-
11 pression in the [input-item](#), nor shall any element appear more than once in a given [input-item](#).

NOTE 3

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

```

- 12 • If an [effective item](#) of derived type in an unformatted input/output statement is not processed by a
13 [defined input/output](#) procedure (12.6.4.8), and if any subobject of that [effective item](#) would be pro-
14 cessed by a [defined input/output](#) procedure, the [effective item](#) is treated as if all of the components
15 of the object were specified in [component order](#) (7.5.4.7); those components shall be accessible in
16 the [scoping unit](#) containing the data transfer statement and shall not be pointers or [allocatable](#).
- 17 • An [effective item](#) of derived type in an unformatted input/output statement is treated as a single
18 value in a processor-dependent form unless the [effective item](#) or a subobject thereof is processed by
19 a [defined input/output](#) procedure (12.6.4.8).

NOTE 4

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.

- 20 • If an [effective item](#) of derived type in a formatted input/output statement is not processed by a
21 [defined input/output](#) procedure, that [effective item](#) is treated as if all of the components of the ef-
22 fective item were specified in [component order](#); those components shall be accessible in the [scoping](#)
23 unit containing the input/output statement and shall not be pointers or [allocatable](#).

- If a derived-type list item is not processed by a [defined input/output](#) procedure and is not treated as a list of its individual components, all the subcomponents of that list item shall be accessible in the [scoping unit](#) containing the data transfer statement and shall not be pointers or [allocatable](#).
- For an *io-implied-do*, the loop initialization and execution are the same as for a [DO construct](#) (11.1.7.4).

NOTE 5

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
```

- 9 The scalar objects resulting when a data transfer statement's list items are expanded according to the rules in this subclause for handling array and derived-type list items are called [effective items](#). Zero-sized arrays and *io-implied-dos* with an iteration count of zero do not contribute to the list of [effective items](#). A scalar character item of zero length is an [effective item](#).

NOTE 6

In a formatted input/output statement, edit descriptors are associated with [effective items](#), which are always scalar. The rules in 12.6.3 determine the set of [effective items](#) corresponding to each actual list item in the statement. These rules might have to be applied repetitively until all of the [effective items](#) are scalar items.

- 10 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](#).

- 11 An output list shall not contain an effective item that is a *boz-literal-constant*.

12.6.4 Execution of a data transfer input/output statement**12.6.4.1 Data transfer sequence of operations**

- 1 Execution of a WRITE or PRINT statement for a unit connected to a file that does not exist creates the file unless an error condition occurs.
- 2 The effect of executing a synchronous data transfer statement shall be as if the following operations were performed in the order specified.
 - (1) Determine the direction of data transfer (12.6.4.2).
 - (2) Identify the [unit](#) (12.6.4.3).
 - (3) Perform a wait operation for all pending input/output operations for the [unit](#). If an error, end-of-file, or end-of-record condition occurs during any of the wait operations, steps 4 through 8 are skipped.
 - (4) Establish the format if one is specified.
 - (5) If the statement is not a child data transfer statement (12.6.4.8),
 - (a) position the file prior to data transfer (12.3.4.3), and
 - (b) for formatted data transfer, set the left tab limit (13.8.1.2).
 - (6) Transfer data between the file and the entities specified by the input/output list (if any) or namelist, possibly mediated by [defined input/output](#) procedures (12.6.4.8).

- (7) Determine whether an error, end-of-file, or end-of-record condition has occurred.
 - (8) Position the file after data transfer (12.3.4.4) unless the statement is a child data transfer statement (12.6.4.8).
 - (9) Cause any variable specified in a SIZE= specifier to become defined.
 - (10) If an error, end-of-file, or end-of-record condition occurred, processing continues as specified in 12.11; otherwise, any variable specified in an IOSTAT= specifier is assigned the value zero.
- 3 The effect of executing an asynchronous data transfer statement shall be as if the following operations were performed in the order specified.
 - (1) Determine the direction of data transfer (12.6.4.2).
 - (2) Identify the unit (12.6.4.3).
 - (3) Optionally, perform wait operations for one or more pending input/output operations for the unit. If an error, end-of-file, or end-of-record condition occurs during any of the wait operations, steps 4 through 9 are skipped.
 - (4) Establish the format if one is specified.
 - (5) Position the file prior to data transfer (12.3.4.3) and, for formatted data transfer, set the left tab limit (13.8.1.2).
 - (6) Establish the set of storage units identified by the input/output list. For an input statement, this might require some or all of the data in the file to be read if an input variable is used as a scalar-int-expr in an io-implied-do-control in the input/output list, as a subscript, substring-range, stride, or is otherwise referenced.
 - (7) Initiate an asynchronous data transfer between the file and the entities specified by the input/output list (if any) or namelist. The asynchronous data transfer may complete (and an error, end-of-file, or end-of-record condition may occur) during the execution of this data transfer statement or during a later wait operation.
 - (8) Determine whether an error, end-of-file, or end-of-record condition has occurred. The conditions may occur during the execution of this data transfer statement or during the corresponding wait operation, but not both.
 - (9) Position the file as if the data transfer had finished (12.3.4.4).
 - (10) Cause any variable specified in a SIZE= specifier to become undefined.
 - (11) If an error, end-of-file, or end-of-record condition occurred, processing continues as specified in 12.11; otherwise, any variable specified in an IOSTAT= specifier is assigned the value zero.
- 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 (12.6.2.5) 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

- 1 executed asynchronous data transfer statement, a wait operation is performed for all pending data transfer
 2 operations on that **unit**. When a condition occurs during a subsequent statement, any actions specified by
 3 **IOSTAT=**, **IOMSG=**, **ERR=**, **END=**, and **EOR=** specifiers for that statement are taken.
- 4 8 If execution of the program is terminated during execution of an **output statement**, the contents of the file
 5 become undefined.

NOTE

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.

6 **12.6.4.2 Direction of data transfer**

- 7 1 Execution of a **READ** statement causes values to be transferred from a file to the entities specified by the in-
 8 put list, if any, or specified within the file itself for **namelist** input. Execution of a **WRITE** or **PRINT** statement
 9 causes values to be transferred to a file from the entities specified by the output list and format specifica-
 10 tion, if any, or by the *namelist-group-name* for **namelist** output.

11 **12.6.4.3 Identifying a unit**

- 12 1 A data transfer statement that contains an input/output control list includes a **UNIT=** specifier that iden-
 13 tifies an **external** or **internal** unit. A **READ** statement that does not contain an input/output control list
 14 specifies a particular processor-dependent **unit**, which is the same as the **unit** identified by * in a **READ**
 15 **statement** that contains an input/output control list (12.5.1) and is the same as the **unit** identified by the
 16 value of the **named constant** **INPUT_UNIT** of the intrinsic module **ISO_FORTRAN_ENV** (16.10.2.13). The
 17 **PRINT statement** specifies some other processor-dependent **unit**, which is the same as the **unit** identified
 18 by * in a **WRITE statement** and is the same as the **unit** identified by the value of the **named constant** **OUT-**
 19 **PUT_UNIT** of the intrinsic module **ISO_FORTRAN_ENV** (16.10.2.24). Thus, each data transfer statement
 20 identifies an **external** or **internal** unit.
- 21 2 The **unit** identified by a data transfer statement shall be **connected** to a file when execution of the statement
 22 begins.

NOTE

The **unit** could be **preconnected**.

23 **12.6.4.4 Establishing a format**

- 24 1 If the input/output control list contains * as a format, list-directed formatting is established. If *namelist-*
 25 *group-name* appears, **namelist** formatting is established. If no **format** or *namelist-group-name* is specified,
 26 unformatted data transfer is established. Otherwise, the format specified by **format** is established.
- 27 2 For output to an **internal file**, a format specification that is in the file or is associated with the file shall not
 28 be specified.
- 29 3 An input list item, or an entity associated with it, shall not contain any portion of an established format
 30 specification.

12.6.4.5 Data transfer

12.6.4.5.1 General

1 Data are transferred between the file and the entities specified by the input/output list or namelist. The list items are processed in the order of the input/output list for all data transfer statements except namelist data transfer statements. The list items for a namelist **input statement** are processed in the order of the entities specified within the input records. The list items for a namelist **output statement** are processed in the order in which the variables are specified in the *namelist-group-object-list*. **Effective items** are derived from the input/output list items as described in 12.6.3.

2 All values needed to determine which entities are specified by an input/output list item are determined at the beginning of the processing of that item.

3 All values are transmitted to or from the entities specified by a list item prior to the processing of any succeeding list item for all data transfer statements.

NOTE

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 (13.11) within the input records are transmitted to the matching entity specified in the *namelist-group-object-list* prior to processing any succeeding entity within the input record for namelist **input statements**. If an entity is specified more than once within the input record during a namelist **input statement**, the last occurrence of the entity specifies the value or values to be used for that entity.

5 If the input/output item is a pointer, data are transferred between the file and the associated **target**.

6 If an **internal file** has been specified, an input/output list item shall not be in the file or associated with the file.

7 During the execution of an **output statement** that specifies an **internal file**, no part of that **internal file** shall be referenced, defined, or become undefined as the result of evaluating any output list item.

8 During the execution of an **input statement** that specifies an **internal file**, no part of that **internal file** shall be defined or become undefined as the result of transferring a value to any input list item.

9 A DO variable becomes defined and its iteration count established at the beginning of processing of the *io-implied-do-object-list* an *io-implied-do*.

10 On output, every entity whose value is to be transferred shall be defined.

12.6.4.5.2 Unformatted data transfer

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**

- 1 **unit** immediately following the current file position.
- 2 4 After each value is transferred, the current file position is moved to a point immediately after the last **file**
3 **storage unit** of the value.
- 4 5 On input from a file **connected** for sequential or direct access, the number of **file storage units** required by
5 the input list shall be less than or equal to the number of **file storage units** in the record.
- 6 6 On input, if the **file storage units** transferred do not contain a value with the same type and type parameters
7 as the input list entity, then the resulting value of the entity is processor dependent except in the following
8 cases.
- 9 • A complex entity may correspond to two real values with the same **kind type parameter** as the com-
10 plex entity.
- 11 • A default character list entity of length n may correspond to n default characters stored in the file,
12 regardless of the length parameters of the entities that were written to these **storage units** of the file.
13 If the file is **connected** for stream input, the characters may have been written by formatted stream
14 output.
- 15 7 On output to a file **connected** for unformatted direct access, the output list shall not specify more values
16 than can fit into the record. If the file is **connected** for direct access and the values specified by the output
17 list do not fill the record, the remainder of the record is undefined.
- 18 8 If the file is **connected** for unformatted sequential access, the record is created with a length sufficient
19 to hold the values from the output list. This length shall be one of the set of allowed record lengths for
20 the file and shall not exceed the value specified in the **RECL= specifier**, if any, of the **OPEN statement** that
21 established the connection.

22 **12.6.4.5.3 Formatted data transfer**

- 23 1 If the file is not **connected** for formatted input/output, formatted data transfer is prohibited.
- 24 2 During formatted data transfer, data are transferred with editing between the file and the entities specified
25 by the input/output list or by the *namelist-group-name*. Format control is initiated and editing is performed
26 as described in Clause 13.
- 27 3 The current record and possibly additional records are read or written.
- 28 4 During advancing input when the pad mode has the value NO, the input list and format specification shall
29 not require more characters from the record than the record contains.
- 30 5 During advancing input when the pad mode has the value YES, blank characters are supplied by the pro-
31 cessor if the input list and format specification require more characters from the record than the record
32 contains.
- 33 6 During nonadvancing input when the pad mode has the value NO, an end-of-record condition (12.11) occurs
34 if the input list and format specification require more characters from the record than the record contains,
35 and the record is complete (12.3.3.4). If the record is incomplete, an end-of-file condition occurs instead of
36 an end-of-record condition.
- 37 7 During nonadvancing input when the pad mode has the value YES, blank characters are supplied by the
38 processor if an **effective item** and its corresponding data edit descriptors require more characters from the

1 record than the record contains. If the record is incomplete, an end-of-file condition occurs; otherwise, an
2 end-of-record condition occurs.

3 8 If the file is [connected](#) for direct access, the record number is increased by one as each succeeding record
4 is read or written.

5 9 On output, if the file is [connected](#) for direct access or is an internal file and the characters specified by the
6 output list and format do not fill a record, blank characters are added to fill the record.

7 10 On output, the output list and format specification shall not specify more characters for a record than have
8 been specified by a [RECL= specifier](#) in the [OPEN statement](#) or the record length of an [internal file](#).

9 **12.6.4.6 List-directed formatting**

10 1 If list-directed formatting has been established, editing is performed as described in [13.10](#).

11 **12.6.4.7 Namelist formatting**

12 1 If namelist formatting has been established, editing is performed as described in [13.11](#).

13 2 Every [allocatable namelist-group-object](#) in the namelist group shall be allocated and every [namelist-group-](#)
14 [object](#) that is a pointer shall be associated with a [target](#). If a [namelist-group-object](#) is [polymorphic](#) or has an
15 [ultimate component](#) that is [allocatable](#) or a pointer, that object shall be processed by a [defined input/output](#)
16 procedure ([12.6.4.8](#)).

17 **12.6.4.8 Defined input/output**

18 **12.6.4.8.1 General**

19 1 [Defined input/output](#) allows a program to override the default handling of derived-type objects and values
20 in data transfer statements described in [12.6.3](#).

21 2 A [defined input/output](#) procedure is a procedure accessible by a [defined-io-generic-spec](#) ([15.4.3.2](#)). A par-
22 ticular [defined input/output](#) procedure is selected as described in [12.6.4.8.4](#).

23 **12.6.4.8.2 Defined input/output procedures**

24 1 For a particular derived type and a particular set of [kind type parameter](#) values, there are four possible sets
25 of [characteristics](#) for [defined input/output](#) procedures; one each for formatted input, formatted output,
26 unformatted input, and unformatted output. The program need not supply all four procedures. The pro-
27 cedures are specified to be used for derived-type input/output by [interface blocks](#) ([15.4.3.2](#)) or by generic
28 [bindings](#) ([7.5.5](#)), with a [defined-io-generic-spec](#) (R1509). The [defined-io-generic-specs](#) for these procedures
29 are [READ \(FORMATTED\)](#), [READ \(UNFORMATTED\)](#), [WRITE \(FORMATTED\)](#), and [WRITE \(UNFORMATTED\)](#),
30 for formatted input, unformatted input, formatted output, and unformatted output respectively.

31 2 In the four [interfaces](#), which specify the [characteristics](#) of [defined input/output](#) procedures, the following
32 syntax term is used:

33 R1221 *dtv-type-spec* **is** TYPE([derived-type-spec](#))
34 **or** CLASS([derived-type-spec](#))

35 C1236 (R1221) If [derived-type-spec](#) specifies an [extensible type](#), the [CLASS](#) keyword shall be used; other-
36 wise, the [TYPE](#) keyword shall be used.

1 C1237 (R1221) All length type parameters of *derived-type-spec* shall be assumed.

2 3 If the *defined-io-generic-spec* is **READ (FORMATTED)**, the *characteristics* shall be the same as those spe-
3 cified by the following *interface*:

```
4      SUBROUTINE my_read_routine_formatted (dtv,          &
5                                          unit,          &
6                                          iotype, v_list, &
7                                          iostat, iomsg)
8
9      ! the derived-type variable
10     dtv-type-spec, INTENT(INOUT) :: dtv
11     INTEGER, INTENT(IN) :: unit ! unit number
12     ! the edit descriptor string
13     CHARACTER (LEN=*), INTENT(IN) :: iotype
14     INTEGER, INTENT(IN) :: v_list(:)
15     INTEGER, INTENT(OUT) :: iostat
16     CHARACTER (LEN=*), INTENT(INOUT) :: iomsg
17     END
```

17 4 If the *defined-io-generic-spec* is **READ (UNFORMATTED)**, the *characteristics* shall be the same as those spe-
18 cified by the following *interface*:

```
19      SUBROUTINE my_read_routine_unformatted (dtv,          &
20                                              unit,          &
21                                              iostat, iomsg)
22
23      ! the derived-type variable
24     dtv-type-spec, INTENT(INOUT) :: dtv
25     INTEGER, INTENT(IN) :: unit
26     INTEGER, INTENT(OUT) :: iostat
27     CHARACTER (LEN=*), INTENT(INOUT) :: iomsg
28     END
```

28 5 If the *defined-io-generic-spec* is **WRITE (FORMATTED)**, the *characteristics* shall be the same as those spe-
29 cified by the following *interface*:

```
30      SUBROUTINE my_write_routine_formatted (dtv,          &
31                                              unit,          &
32                                              iotype, v_list, &
33                                              iostat, iomsg)
34
35      ! the derived-type value/variable
36     dtv-type-spec, INTENT(IN) :: dtv
37     INTEGER, INTENT(IN) :: unit
38     ! the edit descriptor string
39     CHARACTER (LEN=*), INTENT(IN) :: iotype
40     INTEGER, INTENT(IN) :: v_list(:)
41     INTEGER, INTENT(OUT) :: iostat
42     CHARACTER (LEN=*), INTENT(INOUT) :: iomsg
43     END
```

43 6 If the *defined-io-generic-spec* is **WRITE (UNFORMATTED)**, the *characteristics* shall be the same as those
44 specified by the following *interface*:

```
45      SUBROUTINE my_write_routine_unformatted (dtv,          &
46                                              unit,          &
47                                              iostat, iomsg)
48
49      ! the derived-type value/variable
50     dtv-type-spec, INTENT(IN) :: dtv
51     INTEGER, INTENT(IN) :: unit
52     INTEGER, INTENT(OUT) :: iostat
53     CHARACTER (LEN=*), INTENT(INOUT) :: iomsg
54     END
```

54 7 The actual specific procedure names (the `my_..._routine_...` procedure names above) are not significant.
55 In the discussion here and elsewhere, the *dummy arguments* in these *interfaces* are referred to by the
56 names given above; the names are, however, arbitrary.

12.6.4.8.3 Executing defined input/output data transfers

- 1 If a **defined input/output** procedure is selected for an **effective item** as specified in 12.6.4.8.4, the processor shall call the selected **defined input/output** procedure for that **effective item**. The **defined input/output** procedure controls the actual data transfer operations for the **effective 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. As a child data transfer statement and its corresponding parent data transfer statement use the same file connection (12.5), the connection modes at the beginning of execution of the child data transfer statement are those in effect in the parent data transfer statement at the moment when the **defined input/output** procedure was invoked.

NOTE 1

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 2

A particular execution of a READ, WRITE or PRINT statement can be both a parent and a child data transfer statement. A **defined input/output** procedure can indirectly call itself or another **defined input/output** procedure by executing a child data transfer statement containing a list item of derived type, where a matching **interface** is accessible for that derived type. If a **defined input/output** procedure calls itself indirectly in this manner, it cannot be declared **NON_RECURSIVE**.

- 3 A child data transfer statement is processed differently from a nonchild data transfer statement in the following ways.
 - Executing a child data transfer statement does not position the file prior to data transfer.
 - An unformatted child data transfer statement does not position the file after data transfer is complete.
 - Any **ADVANCE= specifier** in a child input/output statement is ignored.
- 4 When a **defined input/output** procedure is invoked, the processor shall pass a **unit** argument that has a value as follows.
 - If the parent data transfer statement uses a **file-unit-number**, the value of the **unit** argument shall be that of the **file-unit-number**.
 - If the parent data transfer statement is a **WRITE statement** with an asterisk **unit** or a **PRINT statement**, the **unit** argument shall have the same value as the **named constant OUTPUT_UNIT** of the intrinsic module **ISO_FORTRAN_ENV** (16.10.2).
 - If the parent data transfer statement is a **READ statement** with an asterisk **unit** or a **READ statement** without an **io-control-spec-list**, the **unit** argument shall have the same value as the **INPUT_UNIT named constant** of the intrinsic module **ISO_FORTRAN_ENV** (16.10.2).
 - Otherwise the parent data transfer statement accesses an **internal file**, in which case the **unit** argument shall have a processor-dependent negative value.

NOTE 3

The `unit` argument passed to a [defined input/output](#) procedure will be negative when the parent data transfer statement specified an [internal unit](#), or specified an [external unit](#) that is a `NEWUNIT` value. When an [internal unit](#) is used with the [INQUIRE statement](#), an error condition will occur, and any variable specified in an `IOSTAT=specifier` will be assigned the value `IOSTAT_INQUIRE_INTERNAL_UNIT` from the intrinsic module `ISO_FORTRAN_ENV` (16.10.2).

- 1 5 For formatted data transfer, the processor shall pass an `iotype` argument that has the value
 - 2 • “LISTDIRECTED” if the parent data transfer statement specified list directed formatting,
 - 3 • “NAMELIST” if the parent data transfer statement specified namelist formatting, or
 - 4 • “DT” concatenated with the [char-literal-constant](#), if any, of the DT edit descriptor in the format specification of the parent data transfer statement.
- 6 6 If the parent data transfer statement is an [input statement](#), the `dtv dummy argument` is argument associated with the [effective item](#) that caused the [defined input](#) procedure to be invoked, as if the [effective item](#) were an [actual argument](#) in this [procedure reference](#) (5.4.5).
- 9 7 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`.
- 11 8 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 4

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.

- 15 9 The `iostat` argument is used to report whether an error, end-of-record, or end-of-file condition (12.11) occurs. If an error condition occurs, the [defined input/output](#) procedure shall assign a positive value to the `iostat` argument. Otherwise, if an end-of-file condition occurs, the [defined input](#) procedure shall assign the value of the [named constant IOSTAT_END](#) (16.10.2.16) to the `iostat` argument. Otherwise, if an end-of-record condition occurs, the [defined input](#) procedure shall assign the value of the [named constant IOSTAT_EOR](#) (16.10.2.17) to `iostat`. Otherwise, the [defined input/output](#) procedure shall assign the value zero to the `iostat` argument.
- 22 10 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 5

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.

- 25 11 If the `iostat` argument of the [defined input/output](#) procedure has a nonzero value when that procedure returns, and the processor therefore [terminates execution](#) of the program as described in 12.11, the processor shall make the value of the `iomsg` argument available in a processor-dependent manner.
- 28 12 While a parent [READ statement](#) is active, an input/output statement shall not read from any [external unit](#) other than the one specified by the `unit dummy argument` and shall not perform output to any [external](#)

- 1 unit.
- 2 13 While a parent **WRITE** or **PRINT** statement is active, an input/output statement shall not perform output
3 to any **external unit** other than the one specified by the `unit` **dummy argument** and shall not read from any
4 **external unit**.
- 5 14 While a parent data transfer statement is active, a data transfer statement that specifies an **internal file** is
6 permitted.
- 7 15 **OPEN**, **CLOSE**, **BACKSPACE**, **ENDFILE**, and **REWIND** statements shall not be executed while a parent data
8 transfer statement is active.
- 9 16 A **defined input/output** procedure may use a format specification with a **DT edit descriptor** for handling a
10 component of the derived type that is itself of a derived type. A child data transfer statement that is a list
11 directed or namelist input/output statement may contain a list item of derived type.
- 12 17 Because a child data transfer statement does not position the file prior to data transfer, the child data trans-
13 fer statement starts transferring data from where the file was positioned by the parent data transfer state-
14 ment's most recently processed **effective item** or edit descriptor. This is not necessarily at the beginning of
15 a record.
- 16 18 The edit descriptors T and TL used on `unit` by a child data transfer statement shall not cause the file to be
17 positioned before the file position at the time the **defined input/output** procedure was invoked.

NOTE 6

A **defined input/output** procedure could use **INQUIRE** to determine the settings of **BLANK=**, **PAD=**, **ROUND=**, **DECIMAL=**, and **DELIM=** for an **external unit**. The **INQUIRE statement** provides values as specified in 12.10.

- 18 19 Neither a parent nor child data transfer statement shall be asynchronous.
- 19 20 A **defined input/output** procedure, and any procedures invoked therefrom, shall not define, nor cause to
20 become undefined, any **storage unit** referenced by any input/output list item, the corresponding format, or
21 any specifier in any active parent data transfer statement, except through the `dtv` argument.

NOTE 7

A data transfer statement with an **ID=**, **POS=**, or **REC=** specifier cannot be a child data transfer statement in a standard-conforming program.

NOTE 8

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

    SUBROUTINE pwf (dtv,unit,iotype,vlist,iostat,iomsg)
      ! argument definitions

```

NOTE 8 (cont.)

```

        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

In the following example, the variables of the derived type `node` form a linked list, with a single value at each node. The subroutine `pwf` is used to write the values in the list, one per line.

```

MODULE p

    TYPE node
        INTEGER :: value = 0
        TYPE (NODE), POINTER :: next_node => NULL ( )
    CONTAINS
        PROCEDURE, PRIVATE :: pwf
        GENERIC             :: WRITE(FORMATTED) => pwf
    END TYPE node

CONTAINS

    SUBROUTINE pwf (dtv,unit,iotype,vlist,iostat,iomsg)
! Write the chain of values, each on a separate line in I9 format.
        CLASS(node), INTENT(IN) :: dtv
        INTEGER, INTENT(IN) :: unit
        CHARACTER (LEN=*), 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

```

12.6.4.8.4 Resolving defined input/output procedure references

1 A suitable **generic interface** for **defined input/output** of an **effective item** is one that has a *defined-io-generic-spec* that is appropriate to the direction (read or write) and form (formatted or unformatted) of the data transfer as specified in 12.6.4.8.2, and has a **specific interface** whose `dtv` argument is compatible with the **effective item** according to the rules for argument association in 15.5.2.5.

2 When an **effective item** (12.6.3) that is of derived type is encountered during a data transfer, **defined input/output** occurs if both of the following conditions are true.

(1) The circumstances of the input/output are such that **defined input/output** is permitted; that is, either

(a) the transfer was initiated by a list-directed, namelist, or unformatted input/output statement, or

(b) a format specification is supplied for the data transfer statement, and the edit descriptor corresponding to the **effective item** is a DT edit descriptor.

(2) A suitable **defined input/output** procedure is available; that is, either

(a) the **declared type** of the **effective item** has a suitable generic **type-bound procedure**, or

(b) a suitable **generic interface** is accessible.

3 If (2a) is true, the procedure referenced is determined as for explicit **type-bound procedure** references (15.5); that is, the **binding** with the appropriate **specific interface** is located in the **declared type** of the **effective item**, and the corresponding **binding** in the **dynamic type** of the **effective item** is selected.

4 If (2a) is false and (2b) is true, the reference is to the procedure identified by the appropriate **specific interface** in the **interface block**.

12.6.5 Termination of data transfer statements

1 Termination of a data transfer statement occurs when

- format processing encounters a colon or data edit descriptor and there are no remaining elements in the *input-item-list* or *output-item-list*,
- unformatted or list-directed data transfer exhausts the *input-item-list* or *output-item-list*,
- namelist output exhausts the *namelist-group-object-list*,
- an error condition occurs,
- an end-of-file condition occurs,
- a slash (/) is encountered as a value separator (13.10, 13.11) in the record being read during list-directed or namelist input, or
- an end-of-record condition occurs during execution of a nonadvancing input statement (12.11).

12.7 Waiting on pending data transfer

12.7.1 Wait operation

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

- 1 transfer operations for the same or multiple **units** simultaneously. A pending data transfer operation re-
 2 mains pending until a corresponding wait operation is performed. A wait operation can be performed by
 3 a **BACKSPACE**, **CLOSE**, **ENDFILE**, **FLUSH**, **INQUIRE**, **PRINT**, **READ**, **REWIND**, **WAIT**, or **WRITE** statement.
- 4 2 A wait operation completes the processing of a pending data transfer operation. Each wait operation com-
 5 pletes only a single data transfer operation, although a single statement may perform multiple wait opera-
 6 tions.
- 7 3 If the actual data transfer is not yet complete, the wait operation first waits for its completion. If the data
 8 transfer operation is an input operation that completed without error, the **storage units** of the **input/output**
 9 **storage sequence** then become defined with the values as described in 12.6.2.16 and 12.6.4.5.
- 10 4 If any error, end-of-file, or end-of-record conditions occur, the applicable actions specified by the **IOSTAT=**,
 11 **IOMSG=**, **ERR=**, **END=**, and **EOR=** specifiers of the statement that performs the wait operation are taken.
- 12 5 If an error or end-of-file condition occurs during a wait operation for a **unit**, the processor performs a wait
 13 operation for all pending data transfer operations for that **unit**.

NOTE

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 raised during a **data transfer statement**, they trigger actions according to the **IOSTAT=**, **ERR=**, **END=**, and **EOR=** specifiers of that statement; if raised during the wait operation, the actions are in accordance with the specifiers of the statement that performs the wait operation.

- 14 6 After completion of the wait operation, the data transfer operation and its input/output storage sequence
 15 are no longer considered to be pending.

16 12.7.2 WAIT statement

- 17 1 A WAIT statement performs a wait operation for specified pending asynchronous data transfer operations.

18 R1222 *wait-stmt* is WAIT (*wait-spec-list*)

19 R1223 *wait-spec* is [UNIT =] *file-unit-number*
 20 or END = *label*
 21 or EOR = *label*
 22 or ERR = *label*
 23 or ID = *scalar-int-expr*
 24 or IOMSG = *iomsg-variable*
 25 or IOSTAT = *stat-variable*

26 C1238 No specifier shall appear more than once in a given *wait-spec-list*.

27 C1239 A *file-unit-number* shall be specified in a *wait-spec-list*; if the optional characters UNIT= are omit-
 28 ted, the *file-unit-number* shall be the first item in the *wait-spec-list*.

29 C1240 (R1223) The *label* in the **ERR=**, **EOR=**, or **END=** specifier shall be the statement label of a **branch**
 30 **target statement** that appears in the same **inclusive scope** as the WAIT statement.

- 31 2 The **IOSTAT=**, **ERR=**, **EOR=**, **END=**, and **IOMSG=** specifiers are described in 12.11.

- 32 3 The value of the expression specified in the ID= specifier shall be zero or the identifier of a pending data
 33 transfer operation for the specified **unit**. If the ID= specifier appears, a wait operation for the specified data

1 transfer operation, if any, is performed. If the ID= specifier is omitted, wait operations for all pending data
2 transfers for the specified **unit** are performed.

3 4 Execution of a WAIT statement specifying a **unit** that does not exist, has no file connected to it, or is not
4 open for asynchronous input/output is permitted, provided that the WAIT statement has no ID= specifier;
5 such a WAIT statement does not cause an error or end-of-file condition to occur.

NOTE

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.

6 12.8 File positioning statements

7 12.8.1 Syntax

8 R1224 *backspace-stmt* **is** BACKSPACE *file-unit-number*
9 **or** BACKSPACE (*position-spec-list*)

10 R1225 *endfile-stmt* **is** ENDFILE *file-unit-number*
11 **or** ENDFILE (*position-spec-list*)

12 R1226 *rewind-stmt* **is** REWIND *file-unit-number*
13 **or** REWIND (*position-spec-list*)

14 1 A **unit** that is connected for direct access shall not be referred to by a BACKSPACE, ENDFILE, or REWIND
15 statement. A **unit** that is connected for unformatted stream access shall not be referred to by a BACKSPACE
16 statement. A **unit** that is connected with an ACTION= specifier having the value READ shall not be referred
17 to by an ENDFILE statement.

18 R1227 *position-spec* **is** [UNIT =] *file-unit-number*
19 **or** IOMSG = *iomsg-variable*
20 **or** IOSTAT = *stat-variable*
21 **or** ERR = *label*

22 C1241 No specifier shall appear more than once in a given *position-spec-list*.

23 C1242 A *file-unit-number* shall be specified in a *position-spec-list*; if the optional characters UNIT= are
24 omitted, the *file-unit-number* shall be the first item in the *position-spec-list*.

25 C1243 (R1227) The *label* in the **ERR= specifier** shall be the statement label of a **branch target statement**
26 that appears in the same **inclusive scope** as the file positioning statement.

27 2 The **IOSTAT=**, **ERR=**, and **IOMSG=** specifiers are described in 12.11.

28 3 Execution of a file positioning statement performs a wait operation for all pending asynchronous data trans-
29 fer operations for the specified **unit**.

30 12.8.2 BACKSPACE statement

31 1 Execution of a BACKSPACE statement causes the file connected to the specified **unit** to be positioned before
32 the current record if there is a current record, or before the preceding record if there is no current record.
33 If the file is at its initial point, the position of the file is not changed.

NOTE 1

If the preceding record is an endfile record, the file is positioned before the endfile record.

- 1 2 If a BACKSPACE statement causes the implicit writing of an endfile record, the file is positioned before the
- 2 record that precedes the endfile record.
- 3 3 Backspacing a file that is connected but does not exist is prohibited.
- 4 4 Backspacing over records written using list-directed or namelist formatting is prohibited.

NOTE 2

An example of a BACKSPACE statement is:

```
BACKSPACE (10, IOSTAT = N)
```

12.8.3 ENDFILE statement

- 6 1 Execution of an ENDFILE statement for a file connected for sequential access writes an endfile record as
- 7 the next record of the file. The file is then positioned after the endfile record, which becomes the last record
- 8 of the file. If the file can also be connected for direct access, only those records before the endfile record are
- 9 considered to have been written. Thus, only those records shall be read during subsequent direct access
- 10 connections to the file.
- 11 2 After execution of an ENDFILE statement for a file connected for sequential access, a **BACKSPACE** or **RE-**
- 12 **WIND** statement shall be used to reposition the file prior to execution of any **data transfer** input/output
- 13 statement or ENDFILE statement.
- 14 3 Execution of an ENDFILE statement for a file connected for stream access causes the terminal point of the
- 15 file to become equal to the current file position. Only **file storage units** before the current position are
- 16 considered to have been written; thus only those **file storage units** shall be subsequently read. Subsequent
- 17 stream **output statements** may be used to write further data to the file.
- 18 4 Execution of an ENDFILE statement for a file that is connected but does not exist creates the file; if the file
- 19 is connected for sequential access, it is created prior to writing the endfile record.

NOTE

An example of an ENDFILE statement is:

```
ENDFILE K
```

12.8.4 REWIND statement

- 20 1 Execution of a REWIND statement causes the specified file to be positioned at its initial point.

NOTE 1

If the file is already positioned at its initial point, execution of this statement has no effect on the position of the file.

- 22 2 Execution of a REWIND statement for a file that is connected but does not exist is permitted and has no
- 23 effect on any file.

NOTE 2

An example of a REWIND statement is:

```
REWIND 10
```

12.9 FLUSH statement

R1228 *flush-stmt* **is** FLUSH *file-unit-number*
or FLUSH (*flush-spec-list*)

R1229 *flush-spec* **is** [UNIT =] *file-unit-number*
or IOSTAT = *stat-variable*
or IOMSG = *iomsg-variable*
or ERR = *label*

C1244 No specifier shall appear more than once in a given *flush-spec-list*.

C1245 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*.

C1246 (R1229) The *label* in the ERR= specifier shall be the statement label of a branch target statement that appears in the same inclusive scope as the FLUSH statement.

1 The IOSTAT=, IOMSG= and ERR= specifiers are described in 12.11.

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 1

Because this document does not specify the mechanism of file storage, the exact meaning of the flush operation is not precisely defined. It is expected that the flush operation will make all data written to a file available to other processes or devices, or make data recently added to a file by other processes or devices available to the program via a subsequent read operation. This is commonly called “flushing input/output buffers”.

NOTE 2

An example of a FLUSH statement is:
 FLUSH (10, IOSTAT = N)

12.10 File inquiry statement

12.10.1 Forms of the INQUIRE statement

1 The INQUIRE statement can be used to inquire about properties of a particular named file, of the connection to a particular unit, or the number of file storage units required for an output list. There are three forms of the INQUIRE statement: inquire by file, which uses the FILE= specifier, inquire by unit, which uses the UNIT= specifier, and inquire by output list, which uses only the IOLENGTH= specifier. Assignments to specifier variables are converted, truncated, or padded according to the rules of 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.

1 3 For inquiry by file, the file specified need not exist or be connected to a **unit**. If it is connected to a **unit**, the
2 inquiry is being made about the connection as well as about the file.

3 4 An INQUIRE statement may be executed before, while, or after a file is connected to a **unit**. All values as-
4 signed by an INQUIRE statement are those that are current at the time the statement is executed.

5 R1230 *inquire-stmt* **is** INQUIRE (*inquire-spec-list*)
6 **or** INQUIRE (IOLENGTH = *scalar-int-variable*) ■
7 ■ *output-item-list*

NOTE

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)
```

8 12.10.2 Inquiry specifiers

9 12.10.2.1 Syntax

10 1 Unless constrained, the following inquiry specifiers may be used in either of the inquire by file or inquire
11 by unit forms of the INQUIRE statement.

12 R1231 *inquire-spec* **is** [UNIT =] *file-unit-number*
13 **or** FILE = *file-name-expr*
14 **or** ACCESS = *scalar-default-char-variable*
15 **or** ACTION = *scalar-default-char-variable*
16 **or** ASYNCHRONOUS = *scalar-default-char-variable*
17 **or** BLANK = *scalar-default-char-variable*
18 **or** DECIMAL = *scalar-default-char-variable*
19 **or** DELIM = *scalar-default-char-variable*
20 **or** DIRECT = *scalar-default-char-variable*
21 **or** ENCODING = *scalar-default-char-variable*
22 **or** ERR = *label*
23 **or** EXIST = *scalar-logical-variable*
24 **or** FORM = *scalar-default-char-variable*
25 **or** FORMATTED = *scalar-default-char-variable*
26 **or** ID = *scalar-int-expr*
27 **or** IOMSG = *iomsg-variable*
28 **or** IOSTAT = *stat-variable*
29 **or** LEADING_ZERO = *scalar-default-char-variable*
30 **or** NAME = *scalar-default-char-variable*
31 **or** NAMED = *scalar-logical-variable*
32 **or** NEXTREC = *scalar-int-variable*
33 **or** NUMBER = *scalar-int-variable*
34 **or** OPENED = *scalar-logical-variable*
35 **or** PAD = *scalar-default-char-variable*
36 **or** PENDING = *scalar-logical-variable*
37 **or** POS = *scalar-int-variable*
38 **or** POSITION = *scalar-default-char-variable*
39 **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*

C1247 No specifier shall appear more than once in a given *inquire-spec-list*.

C1248 An *inquire-spec-list* shall contain one *FILE= specifier* or one *file-unit-number*, but not both.

C1249 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*.

C1250 If an *ID= specifier* appears in an *inquire-spec-list*, a *PENDING= specifier* shall also appear.

C1251 (R1229) The *label* in the *ERR= specifier* shall be the statement label of a *branch target statement* that appears in the same *inclusive scope* as the INQUIRE statement.

2 If *file-unit-number* identifies an *internal unit* (12.6.4.8.2), an error condition occurs.

3 When a returned value of a specifier other than the *NAME= specifier* is of type character, the value returned is in upper case.

4 If an error condition occurs during execution of an INQUIRE statement, all of the inquiry specifier variables become undefined, except for variables in the *IOSTAT=* and *IOMSG=* specifiers (if any).

5 The *IOSTAT=*, *ERR=*, and *IOMSG=* specifiers are described in 12.11.

12.10.2.2 FILE= specifier in the INQUIRE statement

1 The value of the *file-name-expr* in the *FILE= specifier* specifies the name of the file being inquired about. The named file need not exist or be connected to a *unit*. The value of the *file-name-expr* shall be of a form acceptable to the processor as a file name. Any trailing blanks are ignored. The interpretation of case is processor dependent.

12.10.2.3 ACCESS= specifier in the INQUIRE statement

1 The *scalar-default-char-variable* in the *ACCESS= specifier* is assigned the value SEQUENTIAL if the connection is for sequential access, DIRECT if the connection is for direct access, or STREAM if the connection is for stream access. If there is no connection, it is assigned the value UNDEFINED.

12.10.2.4 ACTION= specifier in the INQUIRE statement

1 The *scalar-default-char-variable* in the *ACTION= specifier* is assigned the value READ if the connection is for input only, WRITE if the connection is for output only, and READWRITE if the connection is for both input and output. If there is no connection, the *scalar-default-char-variable* is assigned the value UNDEFINED.

12.10.2.5 ASYNCHRONOUS= specifier in the INQUIRE statement

1 The *scalar-default-char-variable* in the ASYNCHRONOUS= specifier is assigned the value YES if the connection allows asynchronous input/output; it is assigned the value NO if the connection does not allow asynchronous input/output. If there is no connection, the *scalar-default-char-variable* is assigned the value UNDEFINED.

12.10.2.6 BLANK= specifier in the INQUIRE statement

1 The *scalar-default-char-variable* in the BLANK= specifier is assigned the value ZERO or NULL, corresponding to the blank interpretation mode in effect for a connection for formatted input/output. If there is no connection, or if the connection is not for formatted input/output, the *scalar-default-char-variable* is assigned the value UNDEFINED.

12.10.2.7 DECIMAL= specifier in the INQUIRE statement

1 The *scalar-default-char-variable* in the DECIMAL= specifier is assigned the value COMMA or POINT, corresponding to the decimal edit mode in effect for a connection for formatted input/output. If there is no connection, or if the connection is not for formatted input/output, the *scalar-default-char-variable* is assigned the value UNDEFINED.

12.10.2.8 DELIM= specifier in the INQUIRE statement

1 The *scalar-default-char-variable* in the DELIM= specifier is assigned the value APOSTROPHE, QUOTE, or NONE, corresponding to the delimiter mode in effect for a connection for formatted input/output. If there is no connection or if the connection is not for formatted input/output, the *scalar-default-char-variable* is assigned the value UNDEFINED.

12.10.2.9 DIRECT= specifier in the INQUIRE statement

1 The *scalar-default-char-variable* in the DIRECT= specifier is assigned the value YES if DIRECT is included in the set of allowed access methods for the file, NO if DIRECT is not included in the set of allowed access methods for the file, and UNKNOWN if the processor is unable to determine whether DIRECT is included in the set of allowed access methods for the file or if the *unit* identified by *file-unit-number* is not connected to a file.

12.10.2.10 ENCODING= specifier in the INQUIRE statement

1 The *scalar-default-char-variable* in the ENCODING= specifier is assigned the value UTF-8 if the connection is for formatted input/output with an encoding form of UTF-8, and is assigned the value UNDEFINED if the connection is for unformatted input/output. If there is no connection, it is assigned the value UTF-8 if the processor is able to determine that the encoding form of the file is UTF-8; if the processor is unable to determine the encoding form of the file or if the *unit* identified by *file-unit-number* is not connected to a file, the variable is assigned the value UNKNOWN.

NOTE

The value assigned could be something other than UTF-8, UNDEFINED, or UNKNOWN if the processor supports other specific encoding forms (e.g. UTF-16BE).

12.10.2.11 EXIST= specifier in the INQUIRE statement

Execution of an INQUIRE by file statement causes the *scalar-logical-variable* in the EXIST= specifier to be assigned the value true if there exists a file with the specified name; otherwise, false is assigned. Execution of an INQUIRE by unit statement causes true to be assigned if the specified *unit* exists; otherwise, false is assigned.

12.10.2.12 FORM= specifier in the INQUIRE statement

The *scalar-default-char-variable* in the FORM= specifier is assigned the value FORMATTED if the connection is for formatted input/output, and is assigned the value UNFORMATTED if the connection is for unformatted input/output. If there is no connection, it is assigned the value UNDEFINED.

12.10.2.13 FORMATTED= specifier in the INQUIRE statement

The *scalar-default-char-variable* in the FORMATTED= specifier is assigned the value YES if FORMATTED is included in the set of allowed forms for the file, NO if FORMATTED is not included in the set of allowed forms for the file, and UNKNOWN if the processor is unable to determine whether FORMATTED is included in the set of allowed forms for the file or if the *unit* identified by *file-unit-number* is not connected to a file.

12.10.2.14 ID= specifier in the INQUIRE statement

The value of the expression specified in the ID= specifier shall be the identifier of a pending data transfer operation for the specified *unit*. This specifier interacts with the PENDING= specifier (12.10.2.22).

12.10.2.15 LEADING_ZERO= specifier in the INQUIRE statement

The *scalar-default-char-variable* in the LEADING_ZERO= specifier is assigned the value PRINT, SUPPRESS, or PROCESSOR_DEFINED, corresponding to the leading zero mode in effect for a connection for formatted input/output. If there is no connection, or if the connection is not for formatted input/output, the *scalar-default-char-variable* is assigned the value UNDEFINED.

12.10.2.16 NAME= specifier in the INQUIRE statement

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

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.

The case of the characters assigned to *scalar-default-char-variable* is processor dependent.

12.10.2.17 NAMED= specifier in the INQUIRE statement

The *scalar-logical-variable* in the NAMED= specifier is assigned the value true if the file has a name; otherwise, it is assigned the value false.

12.10.2.18 NEXTREC= specifier in the INQUIRE statement

1 The *scalar-int-variable* in the NEXTREC= specifier is assigned the value $n + 1$, where n is the record number of the last record read from or written to the connection for direct access. If there is a connection but no records have been read or written since the connection, the *scalar-int-variable* is assigned the value 1. If there is no connection, the connection is not for direct access, or the position is indeterminate because of a previous error condition, the *scalar-int-variable* becomes undefined. If there are pending data transfer operations for the specified *unit*, the value assigned is computed as if all the pending data transfers had already completed.

12.10.2.19 NUMBER= specifier in the INQUIRE statement

1 Execution of an INQUIRE by file statement causes the *scalar-int-variable* in the NUMBER= specifier to be assigned the value of the *external unit* number of the unit that is connected to the file. If more than one unit on an *image* is connected to the file, which of the connected external unit numbers is assigned to the *scalar-int-variable* is processor dependent. If there is no *unit* connected to the file, the value -1 is assigned. Execution of an INQUIRE by *unit* statement causes the *scalar-int-variable* to be assigned the value of *file-unit-number*.

12.10.2.20 OPENED= specifier in the INQUIRE statement

1 Execution of an INQUIRE by file statement causes the *scalar-logical-variable* in the OPENED= specifier to be assigned the value true if the file specified is connected to a *unit*; otherwise, false is assigned. Execution of an INQUIRE by unit statement causes the *scalar-logical-variable* to be assigned the value true if the specified *unit* is connected to a file; otherwise, false is assigned.

12.10.2.21 PAD= specifier in the INQUIRE statement

1 The *scalar-default-char-variable* in the PAD= specifier is assigned the value YES or NO, corresponding to the pad mode in effect for a connection for formatted input/output. If there is no connection or if the connection is not for formatted input/output, the *scalar-default-char-variable* is assigned the value UNDEFINED.

12.10.2.22 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, no wait operations are performed, and the previously pending data transfers remain pending after the execution of the INQUIRE statement.

NOTE

The processor has considerable flexibility in defining when it considers a transfer to be complete. Any of the following approaches could be used:

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

12.10.2.23 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 are pending data transfer operations for the specified unit, the value assigned is computed as if all the pending data transfers had already completed. If there is no connection, the file is not connected for stream access, or if the position of the file is indeterminate because of previous error conditions, the variable becomes undefined.

12.10.2.24 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.

12.10.2.25 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.

12.10.2.26 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.

12.10.2.27 RECL= specifier in the INQUIRE statement

1 The *scalar-int-variable* in the RECL= specifier is assigned the value of the record length of a connection for direct access, or the value of the maximum record length of a connection for sequential access. If the connection is for formatted input/output, the length is the number of characters for all records that contain

only characters of default kind. If the connection is for unformatted input/output, the length is measured in *file storage units*. If there is no connection, the *scalar-int-variable* is assigned the value -1 , and if the connection is for stream access, the *scalar-int-variable* is assigned the value -2 .

12.10.2.28 ROUND= specifier in the INQUIRE statement

The *scalar-default-char-variable* in the ROUND= specifier is assigned the value UP, DOWN, ZERO, NEAREST, COMPATIBLE, or PROCESSOR_DEFINED, corresponding to the input/output rounding mode in effect for a connection for formatted input/output. If there is no connection or if the connection is not for formatted input/output, the *scalar-default-char-variable* is assigned the value UNDEFINED. The processor shall return the value PROCESSOR_DEFINED only if the behavior of the input/output rounding mode is different from that of the UP, DOWN, ZERO, NEAREST, and COMPATIBLE modes.

12.10.2.29 SEQUENTIAL= specifier in the INQUIRE statement

The *scalar-default-char-variable* in the SEQUENTIAL= specifier is assigned the value YES if SEQUENTIAL is included in the set of allowed access methods for the file, NO if SEQUENTIAL is not included in the set of allowed access methods for the file, and UNKNOWN if the processor is unable to determine whether SEQUENTIAL is included in the set of allowed access methods for the file or if the *unit* identified by *file-unit-number* is not connected to a file.

12.10.2.30 SIGN= specifier in the INQUIRE statement

The *scalar-default-char-variable* in the SIGN= specifier is assigned the value PLUS, SUPPRESS, or PROCESSOR_DEFINED, corresponding to the sign mode in effect for a connection for formatted input/output. If there is no connection, or if the connection is not for formatted input/output, the *scalar-default-char-variable* is assigned the value UNDEFINED.

12.10.2.31 SIZE= specifier in the INQUIRE statement

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 .

For a file that can be connected for stream access, the file size is the number of the highest-numbered *file storage unit* in the file.

For a file that can 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.

If there are pending data transfer operations for the specified unit, the value assigned is computed as if all the pending data transfers had already completed.

12.10.2.32 STREAM= specifier in the INQUIRE statement

The *scalar-default-char-variable* in the STREAM= specifier is assigned the value YES if STREAM is included in the set of allowed access methods for the file, NO if STREAM is not included in the set of allowed access methods for the file, and UNKNOWN if the processor is unable to determine whether STREAM is included in the set of allowed access methods for the file or if the *unit* identified by *file-unit-number* is not connected to a file.

12.10.2.33 UNFORMATTED= specifier in the INQUIRE statement

The *scalar-default-char-variable* in the UNFORMATTED= specifier is assigned the value YES if UNFORMATTED is included in the set of allowed forms for the file, NO if UNFORMATTED is not included in the set of allowed forms for the file, and UNKNOWN if the processor is unable to determine whether UNFORMATTED is included in the set of allowed forms for the file or if the *unit* identified by *file-unit-number* is not connected to a file.

12.10.2.34 WRITE= specifier in the INQUIRE statement

The *scalar-default-char-variable* in the WRITE= specifier is assigned the value YES if WRITE is included in the set of allowed actions for the file, NO if WRITE is not included in the set of allowed actions for the file, and UNKNOWN if the processor is unable to determine whether WRITE is included in the set of allowed actions for the file or if the *unit* identified by *file-unit-number* is not connected to a file.

12.10.3 Inquire by output list

The *scalar-int-variable* in the IOLENGTH= specifier is assigned the processor-dependent number of *file storage units* that would be required to store the data of the output list in an unformatted file. The value shall be suitable as a *RECL= specifier* in an *OPEN statement* that connects a file for unformatted direct access if data will be read from or written to the file using data transfer statements with an input/output list that specifies transfer of a sequence of objects having the same types, type parameters, and extents, in the same order as the output list in the INQUIRE statement.

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 12.6.3. If a derived-type list item appears in the output list, the value returned for the IOLENGTH= specifier assumes that no *defined input/output* procedure will be invoked.

12.11 Error, end-of-record, and end-of-file conditions

12.11.1 Occurrence of input/output conditions

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.

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).

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*.

An end-of-file condition may occur at the beginning of execution of an *input statement*. An end-of-file condition also may occur during execution of a formatted *input statement* when more than one record is required by the interaction of the input list and the format. An end-of-file condition also may occur during execution of a stream *input statement*.

12.11.2 Error conditions and the ERR= specifier

- 1 If an error condition occurs during execution of an input/output statement, the position of the file becomes indeterminate.
- 2 If an error condition occurs during execution of an input/output statement that contains neither an ERR= nor IOSTAT= specifier, **error termination** is initiated. If an error condition occurs during execution of an input/output statement that contains either an ERR= specifier or an IOSTAT= specifier then:
 - (1) processing of the input/output list, if any, terminates;
 - (2) if the statement is a **data transfer statement** or the error condition occurs during a wait operation, all *do-variables* in the statement that initiated the transfer become undefined;
 - (3) if an IOSTAT= specifier appears, the *stat-variable* in the IOSTAT= specifier becomes defined as specified in 12.11.5;
 - (4) if an IOMSG= specifier appears, the *iomsg-variable* becomes defined as specified in 12.11.6;
 - (5) if the statement is a **READ statement** and it contains a SIZE= specifier, the *scalar-int-variable* in the SIZE= specifier becomes defined as specified in 12.6.2.16;
 - (6) if the statement is a **READ statement** or the error condition occurs in a wait operation for a transfer initiated by a **READ statement**, all input items or namelist group objects in the statement that initiated the transfer become undefined;
 - (7) if an ERR= specifier appears, a branch to the statement labeled by the *label* in the ERR= specifier occurs.

12.11.3 End-of-file condition and the END= specifier

- 1 If an end-of-file condition occurs during execution of an input/output statement that contains neither an END= specifier nor an IOSTAT= specifier, **error termination** is initiated. If an end-of-file condition occurs during execution of an input/output statement that contains either an END= specifier or an IOSTAT= specifier, and an error condition does not occur then:
 - (1) processing of the input list, if any, terminates;
 - (2) if the statement is a **data transfer statement** or the end-of-file condition occurs during a wait operation, all *do-variables* in the statement that initiated the transfer become undefined;
 - (3) if the statement is an **input statement** or the end-of-file condition occurs during a wait operation for a transfer initiated by an **input statement**, all **effective items** resulting from the expansion of list items or the namelist group 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 *stat-variable* in the IOSTAT= specifier becomes defined as specified in 12.11.5;
 - (6) if an IOMSG= specifier appears, the *iomsg-variable* becomes defined as specified in 12.11.6;
 - (7) if an END= specifier appears, a branch to the statement labeled by the *label* in the END= specifier occurs.

12.11.4 End-of-record condition and the EOR= specifier

- 1 If an end-of-record condition occurs during execution of an input/output statement that contains neither an EOR= specifier nor an IOSTAT= specifier, **error termination** is initiated. If an end-of-record condition occurs during execution of an input/output statement that contains either an EOR= specifier or an IOSTAT= specifier, and an error condition does not occur then:

- (1) if the pad mode has the value
 - (a) YES, the record is padded with blanks to satisfy the [effective item \(12.6.4.5.3\)](#) and corresponding data edit descriptors that require more characters than the record contains,
 - (b) NO, the [effective item](#) becomes undefined;
- (2) processing of the input list, if any, terminates;
- (3) if the statement is a [data transfer statement](#) or the end-of-record condition occurs during a wait operation, all [do-variables](#) in the statement that initiated the transfer become undefined;
- (4) the file specified in the input statement is positioned after the current record;
- (5) if an [IOSTAT= specifier](#) appears, the [stat-variable](#) in the [IOSTAT= specifier](#) becomes defined as specified in [12.11.5](#);
- (6) if an [IOMSG= specifier](#) appears, the [iomsg-variable](#) becomes defined as specified in [12.11.6](#);
- (7) if a [SIZE= specifier](#) appears, the [scalar-int-variable](#) in the [SIZE= specifier](#) becomes defined as specified in [\(12.6.2.16\)](#);
- (8) if an [EOR= specifier](#) appears, a branch to the statement labeled by the [label](#) in the [EOR= specifier](#) occurs.

12.11.5 IOSTAT= specifier

- 1 Execution of an input/output statement containing the [IOSTAT= specifier](#) causes the [stat-variable](#) in the [IOSTAT= specifier](#) to become defined with
 - a zero value if neither an error condition, an end-of-file condition, nor an end-of-record condition occurs,
 - the processor-dependent positive integer value of the constant [IOSTAT_INQUIRE_INTERNAL_UNIT](#) from the intrinsic module [ISO_FORTRAN_ENV \(16.10.2\)](#) if a [unit](#) number in an [INQUIRE statement](#) identifies an [internal file](#),
 - a processor-dependent positive integer value different from [IOSTAT_INQUIRE_INTERNAL_UNIT](#) if any other error condition occurs,
 - the processor-dependent negative integer value of the constant [IOSTAT_END \(16.10.2.16\)](#) from the intrinsic module [ISO_FORTRAN_ENV](#) if an end-of-file condition occurs and no error condition occurs,
 - the processor-dependent negative integer value of the constant [IOSTAT_EOR \(16.10.2.17\)](#) from the intrinsic module [ISO_FORTRAN_ENV](#) if an end-of-record condition occurs and no error condition or end-of-file condition occurs, or
 - a processor-dependent negative integer value different from [IOSTAT_EOR](#) and [IOSTAT_END](#), if the [IO-STAT= specifier](#) appears in a [FLUSH statement](#) and the processor does not support the flush operation for the specified unit.

NOTE

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

```


12.11.6 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.

12.12 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 value 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 (12.6.3) or *namelist-group-object-list*, nor with a *do-variable* of an *io-implied-do* in the data transfer input/output list.

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 can 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

Restrictions on the evaluation of expressions (10.1.4) prohibit certain side effects.

13 Input/output editing

13.1 Format specifications

- 1 A format used in conjunction with a [data transfer statement](#) provides information that directs the editing between the internal representation of data and the characters of a sequence of formatted records.
- 2 A *format* ([12.6.2.2](#)) in a [data transfer statement](#) can 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 can be an asterisk (*), which indicates list-directed formatting ([13.10](#)). Namelist formatting ([13.11](#)) is indicated by specifying a *namelist-group-name* instead of a *format*.

13.2 Explicit format specification methods

13.2.1 FORMAT statement

R1301 *format-stmt* is `FORMAT format-specification`

R1302 *format-specification* is `( [ format-items ] )`
or `( [ format-items, ] unlimited-format-item )`

C1301 (R1301) The *format-stmt* shall be labeled.

- 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 ([13.9](#)).

NOTE

Examples of FORMAT statements are:

```
5      FORMAT (1PE12.4, I10)
9      FORMAT (I12, /, ' Dates: ', 2 (2I3, I5))
```

13.2.2 Character format specification

- 1 A character expression used as a *format* in a formatted input/output statement shall evaluate to a character string whose leading part is a valid format specification.

NOTE 1

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

1 specification shall be entirely within that array element.

NOTE 2

If a character constant is used as a *format* in *data transfer statement*, care needs to 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 are needed to delimit the edit descriptor and four apostrophes are needed 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.

2 13.3 Form of a format item list

3 13.3.1 Syntax

4 R1303 *format-items* is *format-item* [,] *format-item*] ...

5 R1304 *format-item* is [*r*] *data-edit-desc*
6 or *control-edit-desc*
7 or *char-string-edit-desc*
8 or [*r*] (*format-items*)

9 R1305 *unlimited-format-item* is * (*format-items*)

10 R1306 *r* is *int-literal-constant*

11 C1302 (R1303) The optional comma shall not be omitted except

- 12 • between a P edit descriptor and an immediately following F, E, EN, ES, EX, D, or G edit descriptor
- 13 (13.8.6), possibly preceded by a repeat specification,
- 14 • before a slash edit descriptor when the optional repeat specification does not appear (13.8.2),
- 15 • after a slash edit descriptor, or
- 16 • before or after a colon edit descriptor (13.8.3)

17 C1303 (R1305) An *unlimited-format-item* shall contain at least one data edit descriptor.

18 C1304 (R1306) *r* shall be positive.

19 C1305 (R1306) A kind parameter shall not be specified for *r*.

20 1 The integer literal constant *r* is called a repeat specification.

21 13.3.2 Edit descriptors

22 1 An edit descriptor is a data edit descriptor (*data-edit-desc*), control edit descriptor (*control-edit-desc*), or
23 character string edit descriptor (*char-string-edit-desc*).

- 1 R1307 *data-edit-desc* is I *w* [. *m*]
 2 or B *w* [. *m*]
 3 or O *w* [. *m*]
 4 or Z *w* [. *m*]
 5 or F *w* . *d*
 6 or E *w* . *d* [E *e*]
 7 or EN *w* . *d* [E *e*]
 8 or ES *w* . *d* [E *e*]
 9 or EX *w* . *d* [E *e*]
 10 or G *w* [. *d* [E *e*]]
 11 or L *w*
 12 or A [*w*]
 13 or AT
 14 or D *w* . *d*
 15 or DT [*char-literal-constant*] [(*v-list*)]
- 16 R1308 *w* is *int-literal-constant*
- 17 R1309 *m* is *int-literal-constant*
- 18 R1310 *d* is *int-literal-constant*
- 19 R1311 *e* is *int-literal-constant*
- 20 R1312 *v* is *signed-int-literal-constant*
- 21 C1306 (R1308) *w* shall be zero or positive for the I, B, O, Z, D, E, EN, ES, EX, F, and G edit descriptors. *w*
 22 shall be positive for all other edit descriptors.
- 23 C1307 (R1307) For the G edit descriptor, *d* shall be specified if *w* is not zero.
- 24 C1308 (R1307) For the G edit descriptor, *e* shall not be specified if *w* is zero.
- 25 C1309 (R1307) A kind parameter shall not be specified for the *char-literal-constant* in the DT edit descrip-
 26 tor, or for *w*, *m*, *d*, *e*, and *v*.
- 27 2 An I, B, O, Z, F, E, EN, ES, EX, G, L, A, AT, D, or DT edit descriptor indicates the manner of editing.
- 28 R1313 *control-edit-desc* is *blank-interp-edit-desc*
 29 or *decimal-edit-desc*
 30 or *leading-zero-edit-desc*
 31 or *position-edit-desc*
 32 or *round-edit-desc*
 33 or *sign-edit-desc*
 34 or *k* P
 35 or :
 36 or [*r*] /
- 37 R1314 *k* is *signed-int-literal-constant*
- 38 C1310 (R1314) A kind parameter shall not be specified for *k*.
- 39 3 In *k* P, *k* is called the scale factor.

1 R1315 *position-edit-desc* is T *n*
 2 or TL *n*
 3 or TR *n*
 4 or *n* X

5 R1316 *n* is *int-literal-constant*

6 C1311 (R1316) *n* shall be positive.

7 C1312 (R1316) A kind parameter shall not be specified for *n*.

8 R1317 *blank-interp-edit-desc* is BN
 9 or BZ

10 R1318 *decimal-edit-desc* is DC
 11 or DP

12 R1319 *leading-zero-edit-desc* is LZS
 13 or LZP
 14 or LZ

15 R1320 *round-edit-desc* is RU
 16 or RD
 17 or RZ
 18 or RN
 19 or RC
 20 or RP

21 R1321 *sign-edit-desc* is SS
 22 or SP
 23 or S

24 4 A T, TL, TR, X, slash, colon, SS, SP, S, LZS, LZP, LZ, P, BN, BZ, RU, RD, RZ, RN, RC, RP, DC, or DP edit descriptor
 25 indicates the manner of editing.

26 R1322 *char-string-edit-desc* is *char-literal-constant*

27 C1313 (R1322) A kind parameter shall not be specified for the *char-literal-constant*.

28 5 Each *rep-char* in a character string edit descriptor shall be capable of representation by the processor.

29 6 A character string edit descriptor provides constant data to be output, and is not valid for input.

30 7 The edit descriptors are without regard to case except within a character string edit descriptor.

31 13.3.3 Fields

32 1 A field is a part of a record that is read on input or written on output when format control encounters a data
 33 edit descriptor or a character string edit descriptor. The field width is the size in characters of the field.

34 13.4 Interaction between input/output list and format

35 1 The start of formatted data transfer using a format specification initiates format control (12.6.4.5.3). Each
 36 action of format control depends on information jointly provided by the next edit descriptor in the format

- 1 specification and the next **effective item** in the input/output list, if one exists.
- 2 2 If an input/output list specifies at least one **effective item**, at least one data edit descriptor shall exist in the
- 3 format specification.

NOTE 1

An empty format specification of the form () can be used only if the input/output list has no **effective item** (12.6.4.5). 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.

- 4 3 A format specification is interpreted from left to right. The exceptions are format items preceded by a
- 5 repeat specification *r*, and format reversion (described below).
- 6 4 A format item preceded by a repeat specification is processed as a list of *r* items, each identical to the format
- 7 item but without the repeat specification and separated by commas.

NOTE 2

An omitted repeat specification is treated in the same way as a repeat specification whose value is one.

- 8 5 To each data edit descriptor interpreted in a format specification, there corresponds one **effective item**
- 9 specified by the input/output list (12.6.3), except that an **effective item** of type complex requires the inter-
- 10 pretation of two F, E, EN, ES, EX, D, or G edit descriptors. For each control edit descriptor or character edit
- 11 descriptor, there is no corresponding item specified by the input/output list, and format control commu-
- 12 nicates information directly with the record.
- 13 6 Whenever format control encounters a data edit descriptor in a format specification, it determines whether
- 14 there is a corresponding **effective item** specified by the input/output list. If there is such an item, it trans-
- 15 mits appropriately edited information between the item and the record, and then format control proceeds.
- 16 If there is no such item, format control terminates.
- 17 7 If format control encounters a colon edit descriptor in a format specification and another effective item is
- 18 not specified, format control terminates.
- 19 8 If format control encounters the rightmost parenthesis of an unlimited format item, control reverts to the
- 20 leftmost parenthesis of that unlimited format item. This reversion of format control has no effect on the
- 21 changeable modes (12.5.2).
- 22 9 If format control encounters the rightmost parenthesis of a complete format specification and another
- 23 effective item is not specified, format control terminates. However, if another effective item is specified,
- 24 format control then reverts to the beginning of the format item terminated by the last preceding right par-
- 25 entthesis that is not part of a DT edit descriptor. If there is no such preceding right parenthesis, format
- 26 control reverts to the first left parenthesis of the format specification. If any reversion occurs, the reused
- 27 portion of the format specification shall contain at least one data edit descriptor. If format control reverts
- 28 to a parenthesis that is preceded by a repeat specification, the repeat specification is reused. Reversion of
- 29 format control, of itself, has no effect on the **changeable modes**. The file is positioned in a manner identical
- 30 to the way it is positioned when a slash edit descriptor is processed (13.8.2).

NOTE 3

Example: The format specification:

```
10 FORMAT (1X, 2(F10.3, I5))
```

with the output statement

```
WRITE (10,10) 10.1, 3, 4.7, 1, 12.4, 5, 5.2, 6
```

NOTE 3 (cont.)

produces the same output as the format specification:

```
10 FORMAT (1X, F10.3, I5, F10.3, I5/F10.3, I5, F10.3, I5)
```

NOTE 4

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.

13.5 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 13.8.1.1. After each slash edit descriptor is processed, the file is positioned as described in 13.8.2.
- 3 During formatted stream output, processing of an A or AT edit descriptor can cause file positioning to occur (13.7.4).
- 4 If format control reverts as described in 13.4, the file is positioned in a manner identical to the way it is positioned when a slash edit descriptor is processed (13.8.2).
- 5 During a read operation, any unprocessed characters of the current record are skipped whenever the next record is read.

13.6 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.

13.7 Data edit descriptors

13.7.1 Purpose of data edit descriptors

- 1 A data edit descriptor causes the conversion of data to or from its internal representation; during formatted stream output, an A or AT data edit descriptor can 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 [effective 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.

13.7.2 Numeric editing

13.7.2.1 General rules

1 The I, B, O, Z, F, E, EN, ES, EX, D, and G edit descriptors can be used to specify the input/output of integer, real, and complex data. The I, B, O, Z and G edit descriptors can be used to specify the input/output of enum type data. The I, B, O, and Z edit descriptors can be used to specify input/output of enumeration type 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-significant number ([13.7.2.3.2](#)), the interpretation of blanks, other than leading blanks, is determined by the blank interpretation mode ([13.8.7](#)). 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 nonnegative 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

When the sign mode is PLUS, a plus sign is not optional.

- (6) On output, with I, B, O, Z, D, E, EN, ES, EX, F, and G editing, the specified value of the field width *w* may be zero. In such cases, the processor selects the smallest positive actual field width that does not result in a field filled with asterisks. The specified value of *w* shall not be zero on input.
- (7) On output of a real zero value, the digits in the exponent field shall all be zero.

13.7.2.2 Integer editing

- 1 The *Iw* and *Iw.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 *effective item* shall be of type integer or of enum or enumeration type. The G, B, O, and Z edit descriptors also may be used to edit integer data (13.7.5.2.2, 13.7.2.4).
- 2 On input, *m* has no effect.
- 3 In the standard form of the input field for the I edit descriptor, the character string is a *signed-digit-string* (R710), except for the interpretation of blanks. If the input field does not have the standard form and is not acceptable to the processor, an error condition occurs.
- 4 The output field for the *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

A *digit-string* always consists of at least one digit.

- 5 The output field for the *Iw.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.
- 6 If the *effective item* for output is of enumeration type, the value output is its ordinal position. If the *effective item* for input is of enumeration type, the value of the input field shall be positive and less than or equal to the number of enumerators; the value assigned to the *effective item* is the enumeration value with that ordinal position.
- 7 If the *effective item* for output is of enum type, the value output is its corresponding integer value. If the *effective item* for input is of enum type, the value assigned is the enum value corresponding to the value of the input field.

13.7.2.3 Real and complex editing

13.7.2.3.1 General

- 1 The F, E, EN, ES, EX, and D edit descriptors specify the editing of real and complex data. An *effective item* corresponding to an F, E, EN, ES, EX, or D edit descriptor shall be real or complex. The G, B, O, and Z edit descriptors also may be used to edit real and complex data (13.7.5.2.3, 13.7.2.4).

13.7.2.3.2 F editing

- 1 The *Fw.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

- 1 not be zero.
- 2 2 A lower-case letter is equivalent to the corresponding upper-case letter in an IEEE exceptional specification
3 or the exponent in a numeric input field.
- 4 3 The standard form of the input field is an IEEE exceptional specification, a hexadecimal-significand number,
5 or consists of a mantissa optionally followed by an exponent. The form of the mantissa is an optional sign,
6 followed by a string of one or more digits optionally containing a *decimal symbol*, including any blanks in-
7 terpreted as zeros. The *d* has no effect on input if the input field contains a *decimal symbol*. If the *decimal*
8 *symbol* is omitted, the rightmost *d* digits of the string, with leading zeros assumed if necessary, are inter-
9 preted as the fractional part of the value represented. The string of digits may contain more digits than a
10 processor uses to approximate the value. The form of the exponent is one of the following:
- 11 • a *sign* followed by a *digit-string*;
 - 12 • the letter E followed by zero or more blanks, followed by a *signed-digit-string*;
 - 13 • the letter D followed by zero or more blanks, followed by a *signed-digit-string*.
- 14 4 An exponent containing a D is processed identically to an exponent containing an E.

NOTE 1

If the input field does not contain an exponent, the effect is as if the basic form were followed by an exponent with a value of $-k$, where k is the established scale factor (13.8.6).

- 15 5 An input field that is an IEEE exceptional specification consists of optional blanks, followed by either
16 • an optional sign, followed by the string 'INF' or the string 'INFINITY', or
17 • an optional sign, followed by the string 'NaN', optionally followed by zero or more alphanumeric
18 characters enclosed in parentheses,
19 optionally followed by blanks.
- 20 6 The value specified by 'INF' or 'INFINITY' is an IEEE infinity; this form shall not be used if the processor
21 does not support IEEE infinities for the input variable. The value specified by 'NaN' is an *IEEE NaN*; this
22 form shall not be used if the processor does not support *IEEE NaNs* for the input variable. The *NaN* value
23 is a quiet *NaN* if the only nonblank characters in the field are 'NaN' or 'NaN()'; otherwise, the *NaN* value is
24 processor dependent. The interpretation of a sign in a *NaN* input field is processor dependent.
- 25 7 An input field that is a hexadecimal-significand number consists of an optional sign, followed by the hexa-
26 decimal indicator which is the digit 0 immediately followed by the letter X, followed by a hexadecimal sig-
27 nificand followed by a hexadecimal exponent. A hexadecimal significand is a string of one or more hexa-
28 decimal characters optionally containing a decimal symbol. The decimal symbol indicates the position
29 of the hexadecimal point; if no decimal symbol appears, the hexadecimal point implicitly follows the last
30 hexadecimal symbol. A hexadecimal exponent is the letter P followed by a (decimal) *signed-digit-string*.
31 Embedded blanks are not permitted in a hexadecimal-significand number; trailing blanks are ignored. The
32 value is equal to the significand multiplied by two raised to the power of the exponent, negated if the op-
33 tional sign is minus.
- 34 8 If the input field does not have one of the standard forms, and is not acceptable to the processor, an error
35 condition occurs.
- 36 9 For an internal value that is an IEEE infinity, the output field consists of blanks, if necessary, followed by a
37 minus sign for negative infinity or an optional plus sign otherwise, followed by the letters 'Inf' or 'Infinity',

right justified within the field. The minimum field width required for output of the form 'Inf' is 3 if no sign is produced, and 4 otherwise. The minimum field width required for output of the form 'Infinity' is 8 if no sign is produced, and 9 otherwise. If w is greater than or equal to the minimum required for the form 'Infinity', the form 'Infinity' is output. If w is zero or w is less than the minimum required for the form 'Infinity' and greater than or equal to the minimum required for the form 'Inf', the form 'Inf' is output. Otherwise (w is greater than zero but less than the minimum required for any form), the field is filled with asterisks.

For an internal value that is an IEEE NaN, the output field consists of blanks, if necessary, followed by the letters 'NaN' and optionally followed by one to $w-5$ alphanumeric processor-dependent characters enclosed in parentheses, right justified within the field. If w is greater than zero and less than 3, the field is filled with asterisks. If w is zero, the output field is 'NaN'.

NOTE 2

The processor-dependent characters following 'NaN' might convey additional information about that particular NaN.

For an internal value that is neither an IEEE infinity nor a NaN, the output field consists of blanks, if necessary, followed by a minus sign if the internal value is negative, or an optional plus sign otherwise, followed by a string of digits that contains a decimal symbol and represents the magnitude of the internal value, as modified by the established scale factor and rounded (13.7.2.3.8) to d fractional digits. Leading zeros are not permitted except for an optional zero immediately to the left of the decimal symbol if the magnitude of the value in the output field is less than one. The optional zero shall appear if there would otherwise be no digits in the output field.

13.7.2.3.3 E and D editing

The $Ew.d$, $Dw.d$, and $Ew.d Ee$ 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.

The form and interpretation of the input field is the same as for $Fw.d$ editing (13.7.2.3.2).

For an internal value that is an IEEE infinity or NaN, the form of the output field is the same as for $Fw.d$.

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

$$[\pm][0].x_1x_2\dots x_dexp$$

where:

- $\pm$ signifies a plus sign or a minus sign;
- $.$ signifies a decimal symbol (13.6);
- $x_1x_2\dots x_d$ are the d most significant digits of the internal value after rounding (13.7.2.3.8);
- exp is a decimal exponent having one of the forms specified in Table 13.1.

Table 13.1 — E and D exponent forms

Edit Descriptor	Absolute Value of Exponent	Form of Exponent ¹
$Ew.d$ with $w > 0$	$ exp \leq 99$	$E\pm z_1z_2$ or $\pm 0z_1z_2$
	$99 < exp \leq 999$	$\pm z_1z_2z_3$

E and D exponent forms (cont.)

Edit Descriptor	Absolute Value of Exponent	Form of Exponent ¹
E <i>w.d</i> Ee with <i>e</i> > 0	$ exp \leq 10^e - 1$	$E \pm z_1 z_2 \dots z_e$
E <i>w.d</i> E0 or E0.d	any	$E \pm z_1 z_2 \dots z_s$
D <i>w.d</i> with <i>w</i> > 0	$ exp \leq 99$	$D \pm z_1 z_2$ or $E \pm z_1 z_2$ or $\pm 0 z_1 z_2$
	$99 < exp \leq 999$	$\pm z_1 z_2 z_3$
D0.d	any	$D \pm z_1 z_2 \dots z_s$ or $E \pm z_1 z_2 \dots z_s$
(1) where each <i>z</i> is a digit, and <i>s</i> is the minimum number of digits required to represent the exponent. A plus sign is produced if the exponent value is zero.		

1 5 The scale factor *k* controls the decimal normalization (13.3.2, 13.8.6). If $-d < k \leq 0$, the output field
2 contains exactly $|k|$ leading zeros and $d - |k|$ significant digits after the decimal symbol. If $0 < k < d + 2$,
3 the output field contains exactly *k* significant digits to the left of the decimal symbol and $d - k + 1$ significant
4 digits to the right of the decimal symbol. Other values of *k* are not permitted.

5 13.7.2.3.4 EN editing

6 1 The EN edit descriptor produces an output field in the form of a real number in engineering notation such
7 that the decimal exponent is divisible by three and the absolute value of the significand (R715) is greater
8 than or equal to 1 and less than 1000, except when the output value is zero. The scale factor has no effect
9 on output.

10 2 The forms of the edit descriptor are **EN***w.d* and **EN***w.d* **Ee** indicating that the external field occupies *w* pos-
11 itions, except when *w* is zero in which case the processor selects the field width. The fractional part of the
12 field contains *d* digits. If *e* is positive the exponent part contains *e* digits, otherwise it contains the minimum
13 number of digits required to represent the exponent value.

14 3 The form and interpretation of the input field is the same as for **F***w.d* editing (13.7.2.3.2).

15 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*.

16 5 For an internal value that is neither an IEEE infinity nor a NaN, the form of the output field is

17 $[\pm]yyy . x_1 x_2 \dots x_d exp$

18 where:

- 19 • $\pm$ signifies a plus sign or a minus sign;
- 20 • *yyy* are the 1 to 3 decimal digits representative of the most significant digits of the internal value after
21 rounding (13.7.2.3.8);
- 22 • *yyy* is an integer such that $1 \leq yyy < 1000$ or, if the output value is zero, *yyy* = 0;
- 23 • . signifies a decimal symbol (13.6);
- 24 • $x_1 x_2 \dots x_d$ are the *d* next most significant digits of the internal value after rounding;
- 25 • *exp* is a decimal exponent, divisible by three, having one of the forms specified in Table 13.2.

Table 13.2 — EN exponent forms

Edit Descriptor	Absolute Value of Exponent	Form of Exponent ¹
EN $w.d$ with $w > 0$	$ 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$	$E\pm z_1 z_2 \dots z_e$
EN $w.d$ E0 or EN0. d	any	$E\pm z_1 z_2 \dots z_s$

(1) where each z is a digit, and s is the minimum number of digits required to represent the exponent. A plus sign is produced if the exponent value is zero.

NOTE

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

13.7.2.3.5 ES editing

1 The ES edit descriptor produces an output field in the form of a real number in scientific notation such that the absolute value of the significand (R715) is greater than or equal to 1 and less than 10, except when the output value is zero. The scale factor has no effect on output.

2 The forms of the edit descriptor are ES $w.d$ and ES $w.d$ 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 (13.7.2.3.2).

4 For an internal value that is an IEEE infinity or NaN, the form of the output field is the same as for F $w.d$.

5 For an internal value that is neither an IEEE infinity nor a NaN, the form of the output field is

$$[\pm]y . x_1 x_2 \dots x_d exp$$

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 (13.7.2.3.8);
- $.$ signifies a decimal symbol (13.6);
- $x_1 x_2 \dots x_d$ are the d next most significant digits of the internal value after rounding;
- exp is a decimal exponent having one of the forms specified in Table 13.3.

Table 13.3 — ES exponent forms

Edit Descriptor	Absolute Value of Exponent	Form of Exponent ¹
ES $w.d$ with $w > 0$	$ exp \leq 99$	$E\pm z_1 z_2$ or $\pm 0 z_1 z_2$
	$99 < exp \leq 999$	$\pm z_1 z_2 z_3$
ES $w.d$ Ee with $e > 0$	$ exp \leq 10^e - 1$	$E\pm z_1 z_2 \dots z_e$
ES $w.d$ E0 or ES0. d	any	$E\pm z_1 z_2 \dots z_s$

ES 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. A plus sign is produced if the exponent value is zero.		

NOTE

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

13.7.2.3.6 EX editing

1 The EX edit descriptor produces an output field in the form of a hexadecimal-significand number.

2 The EX $w.d$ and EX $w.dEe$ edit descriptors indicate that the external field occupies w positions, except when w is zero in which case the processor selects the field width. The fractional part of the field contains d hexadecimal digits, except when d is zero in which case the processor selects the number of hexadecimal digits to be the minimum required so that the output field is equal to the internal value; d shall not be zero if the radix of the internal value is not a power of two. The hexadecimal point, represented by a decimal symbol, appears after the first hexadecimal digit. For the form EX $w.d$, and for EX $w.dE0$, the exponent part contains the minimum number of digits needed to represent the exponent; otherwise the exponent contains e digits. The e has no effect on input. The scale factor has no effect on output.

3 The form and interpretation of the input field is the same as for F $w.d$ editing (13.7.2.3.2).

4 For an internal value that is an IEEE infinity or NaN, the form of the output field is the same as for F $w.d$.

5 For an internal value that is neither an IEEE infinity nor a NaN, the form of the output field is

[±] 0X $x_0 . x_1 x_2 \dots exp$

where:

- $\pm$ signifies a plus sign or a minus sign;
- $.$ signifies a decimal symbol (13.6);
- $x_0 x_1 x_2 \dots$ are the most significant hexadecimal digits of the internal value, after rounding if d is not zero (13.7.2.3.8);
- exp is a binary exponent expressed as a decimal integer; for EX $w.d$ and EX $w.dE0$, the form of the exponent is $P \pm z_1 \dots z_n$, where n is the minimum number of digits needed to represent exp , and for EX $w.dEe$ with e greater than zero the form is $P \pm z_1 \dots z_e$. The choice of binary exponent is processor dependent. If the most significant binary digits of the internal value are $b_0 b_1 b_2 \dots$, the binary exponent might make the value of x_0 be that of b_0 , $b_0 b_1$, $b_0 b_1 b_2$, or $b_0 b_1 b_2 b_3$. A plus sign is produced if the exponent value is zero.

NOTE

Examples:

NOTE (cont.)

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	0X1.00004P+20
2.375	EX0.1	0X2.6P+0

13.7.2.3.7 Complex editing

A complex datum consists of a pair of separate real data. The editing of a scalar datum of complex type is specified by two edit descriptors each of which specifies the editing of real data. The first edit descriptor specifies the editing for the real part; the second specifies it for the imaginary part. The two edit descriptors may be different. Control and character string edit descriptors may be processed between the edit descriptor for the real part and the edit descriptor for the imaginary part.

13.7.2.3.8 Input/output rounding mode

The input/output rounding mode can be specified by an [OPEN statement \(12.5.2\)](#), a [data transfer statement \(12.6.2.14\)](#), or an edit descriptor ([13.8.8](#)).

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.

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.

On processors that support IEEE rounding on conversions ([17.4](#)), NEAREST shall correspond to round to nearest, as specified in ISO/IEC 60559:2020.

NOTE

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 nearest 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 60559:2020 for round toward $+\infty$, round toward $-\infty$, and round toward zero, respectively.

13.7.2.4 B, O, and Z editing

The [Bw](#), [Bw.m](#), [Ow](#), [Ow.m](#), [Zw](#), and [Zw.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

1 not be zero. The corresponding *effective item* shall be of type integer, real, or complex, or of enum or
2 enumeration type.

3 2 On input, *m* has no effect.

4 3 In the standard form of the input field for the B, O, and Z edit descriptors the character string consists of
5 binary, octal, or hexadecimal digits (as in R774, R775, R776) in the respective input field. The lower-case
6 hexadecimal digits a through f in a hexadecimal input field are equivalent to the corresponding upper-case
7 hexadecimal digits. If the input field does not have the standard form, and is not acceptable to the processor,
8 an error condition occurs.

9 4 Input editing produces the value INT (X) if the *effective item* is of type integer and REAL (X) if the *effective*
10 *item* is of type real or complex, where X is a *boz-literal-constant* that specifies the same bit sequence as the
11 digits of the input field. If the *effective item* is of enum or enumeration type ET, the value is ET (INT (X)).

12 5 The output field for the B*w*, O*w*, and Z*w* descriptors consists of zero or more leading blanks followed by
13 the internal value in a form identical to the digits of a binary, octal, or hexadecimal constant, respectively,
14 that specifies the same bit sequence but without leading zero bits.

NOTE

A binary, octal, or hexadecimal constant always consists of at least one digit or hexadecimal digit.

15 R1323 *hex-digit-string* is *hex-digit* [*hex-digit*] ...

16 6 The output field for the B*w.m*, O*w.m*, and Z*w.m* edit descriptor is the same as for the B*w*, O*w*, and Z*w* edit
17 descriptor, except that the *digit-string* or *hex-digit-string* consists of at least *m* digits. If necessary, sufficient
18 leading zeros are included to achieve the minimum of *m* digits. The value of *m* shall not exceed the value of
19 *w*, except when *w* is zero. If *m* is zero and the internal value consists of all zero bits, the output field consists
20 of only blank characters. When *m* and *w* are both zero, and the internal value consists of all zero bits, one
21 blank character is produced.

13.7.3 Logical editing

23 1 The L*w* edit descriptor indicates that the field occupies *w* positions. The corresponding *effective item* shall
24 be of type logical. The G edit descriptor also may be used to edit logical data (13.7.5.3).

25 2 The standard form of the input field consists of optional blanks, optionally followed by a period, followed
26 by a T for true or F for false. The T or F may be followed by additional characters in the field, which are
27 ignored. If the input field does not have the standard form, and is not acceptable to the processor, an error
28 condition occurs.

29 3 A lower-case letter is equivalent to the corresponding upper-case letter in a logical input field.

NOTE

The logical constants .TRUE. and .FALSE. are acceptable input forms.

30 4 The output field consists of *w*−1 blanks followed by a T or F, depending on whether the internal value is
31 true or false, respectively.

13.7.4 Character editing

- 1 The A[*w*] edit descriptor is used with an *effective item* of type character. The AT edit descriptor is used with an *effective item* of type character in an *output statement*; it shall not be used for input. The G edit descriptor also may be used to edit character data (13.7.5.4). The kind type parameter of all characters transferred and converted under control of one A, AT, or G edit descriptor is implied by the kind of the corresponding *effective 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 *effective item*, regardless of the value of the kind type parameter.
- 3 Let *len* be the length of the *effective item*. If the specified field width *w* for an A edit descriptor corresponding to an *effective item* on input 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 *effective item* on output 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.
- 5 The field width for an AT edit descriptor is the length of the value of the *effective item* after any trailing blanks are removed. The output field consists of the value of the *effective item* after any trailing blanks are removed; if the value of the *effective item* is all blanks, no output is produced by the edit descriptor.

NOTE 1

For nondefault character kinds, the blank padding character is processor dependent.

- 6 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 2

If the intrinsic function `NEW_LINE` returns a blank character for a particular character kind, then the processor does not support using a character of that kind to cause record termination in a formatted stream file.

13.7.5 Generalized editing

13.7.5.1 Overview

- 1 The G*w*, G*w.d* and G*w.d* E*e* edit descriptors are used with an *effective item* of enum type or any intrinsic type. When *w* is nonzero, these edit descriptors indicate that the external field occupies *w* positions. For real or complex data the fractional part consists of a maximum of *d* digits and the exponent part consists of *e* digits. When these edit descriptors are used to specify the input/output of integer, logical, or character data, *d* and *e* have no effect. When *w* is zero the processor selects the field width. On input, *w* shall not be zero.

13.7.5.2 Generalized numeric editing

13.7.5.2.1 Overview

- 1 When used to specify the input/output of integer, real, complex, and enum data, the *Gw*, *Gw.d* and *Gw.d Ee* edit descriptors follow the general rules for numeric editing (13.7.2).

NOTE

The *Gw.d Ee* edit descriptor follows any additional rules for the *Ew.d Ee* edit descriptor.

13.7.5.2.2 Generalized integer and enum editing

- 1 When used to specify the input/output of integer or enum data, the *Gw*, *Gw.d*, and *Gw.d Ee* edit descriptors follow the rules for the *Iw* edit descriptor (13.7.2.2). Note that *w* cannot be zero for input editing (13.7.5.1).

13.7.5.2.3 Generalized real and complex editing

- 1 The form and interpretation of the input field for *Gw.d* and *Gw.d Ee* editing is the same as for *Fw.d* editing (13.7.2.3.2). The rest of this subclause applies only to output editing.

- 2 If *w* is nonzero and *d* is zero, *kPEw.0* or *kPEw.0Ee* editing is used for *Gw.0* editing or *Gw.0Ee* editing respectively.

- 3 When used to specify the output of real or complex data that is not an IEEE infinity or NaN, the *G0* and *G0.d* edit descriptors follow the rules for the *Gw.dEe* edit descriptor, except that any leading or trailing blanks are removed. Reasonable processor-dependent values of *w*, *d* (if not specified), and *e* are used with each output value.

- 4 For an internal value that is an IEEE infinity or NaN, the form of the output field for the *Gw.d* and *Gw.d Ee* edit descriptors is the same as for *Fw.d*, and the form of the output field for the *G0* and *G0.d* edit descriptors is the same as for *F0.0*.

- 5 Otherwise, the method of representation in the output field depends on the magnitude of the internal value being edited. If the internal value is zero, let *s* be one. If the internal value is a number other than zero, let *N* be the decimal value that is the result of converting the internal value to *d* significant digits according to the input/output rounding mode and let *s* be the integer such that $10^{s-1} \leq |N| < 10^s$. If $s < 0$ or $s > d$, *kPEw.d* or *kPEw.dEe* editing is used for *Gw.d* editing or *Gw.dEe* editing respectively, where *k* is the scale factor (13.8.6). 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 *Gw.d* editing, *e + 2* for *Gw.dEe* editing if *e* > 0, and 4 for *Gw.dE0* editing.

- 6 The value of *w - n* shall be positive.

NOTE

The scale factor has no effect on output unless the magnitude of the datum to be edited is outside the range that permits effective use of F editing.

13.7.5.3 Generalized logical editing

- 1 When used to specify the input/output of logical data, the *Gw.d* and *Gw.d Ee* edit descriptors with nonzero *w* follow the rules for the *Lw* edit descriptor (13.7.3). When used to specify the output of logical data, the *G0* and *G0.d* edit descriptors follow the rules for the *L1* edit descriptor.

13.7.5.4 Generalized character editing

- 1 When used to specify the input/output of character data, the `Gw.d` and `Gw.d Ee` edit descriptors with non-zero `w` follow the rules for the `Aw` edit descriptor (13.7.4). When used to specify the output of character data, the `G0` and `G0.d` edit descriptors follow the rules for the `A` edit descriptor with no field width.

13.7.6 User-defined derived-type editing

- 1 The DT edit descriptor specifies that a user-provided procedure shall be used instead of the processor's default input/output formatting for processing an *effective item* of derived type.
- 2 The DT edit descriptor may include a character literal constant. The character value "DT" concatenated with the character literal constant is passed to the *defined input/output* procedure as the `iotype` argument (12.6.4.8). The `v` values of the edit descriptor are passed to the *defined input/output* procedure as the `v_list` array argument.

NOTE

For the edit descriptor <code>DT'Link List'(10, 4, 2)</code> , <code>iotype</code> is "DTLink List" and <code>v_list</code> is [10, 4, 2].

- 3 If a derived-type variable or value corresponds to a DT edit descriptor, there shall be an accessible *interface* to a corresponding *defined input/output* procedure for that derived type (12.6.4.8). A DT edit descriptor shall not correspond to an *effective item* that is not of a derived type.

13.8 Control edit descriptors

13.8.1 Position edit descriptors

13.8.1.1 Position editing

- 1 The position edit descriptors T, TL, TR, and X, specify the position at which the next character will be transmitted to or from the record. If any character skipped by a position edit descriptor is of type nondefault character, and the *unit* is a default character *internal file* or an *external* non-Unicode file, the result of that position editing is processor dependent.
- 2 On input, if the position specified by a position edit descriptor is before the current position, portions of a record can be processed more than once, possibly with different editing.
- 3 On input, a position beyond the last character of the record may be specified if no characters are transmitted from such positions.
- 4 On output, a position edit descriptor does not by itself cause characters to be transmitted and therefore does not by itself affect the length of the record. If characters are transmitted to positions at or after the position specified by a position edit descriptor, positions skipped and not previously filled are filled with blanks. The result is as if the entire record were initially filled with blanks.
- 5 On output, a character in the record can be replaced. A position edit descriptor never directly causes a character already placed in the record to be replaced, but it might result in positioning such that subsequent editing causes a replacement.

13.8.1.2 T, TL, and TR editing

1 The left tab limit affects file positioning by the T and TL edit descriptors. Immediately prior to nonchild data transfer (12.6.4.8.3), the left tab limit becomes defined as the character position of the current record or the current position of the stream file. If, during data transfer, the file is positioned to another record, the left tab limit becomes defined as character position one of that record.

2 The T_n edit descriptor indicates that the transmission of the next character to or from a record is to occur at the _nth character position of the record, relative to the left tab limit. This position can be in either direction from the current position.

3 The TL_n edit descriptor indicates that the transmission of the next character to or from the record is to occur at the character position _n characters backward from the current position. However, if _n is greater than the difference between the current position and the left tab limit, the TL_n edit descriptor indicates that the transmission of the next character to or from the record is to occur at the left tab limit.

4 The TR_n edit descriptor indicates that the transmission of the next character to or from the record is to occur at the character position _n characters forward from the current position.

13.8.1.3 X editing

1 The _nX edit descriptor indicates that the transmission of the next character to or from a record is to occur at the character position _n characters forward from the current position.

NOTE

An _nX edit descriptor has the same effect as a TR_n edit descriptor.

13.8.2 Slash editing

1 The slash edit descriptor indicates the end of data transfer to or from the current record.

2 On input from a file connected for sequential or stream access, the remaining portion of the current record is skipped and the file is positioned at the beginning of the next record. This record becomes the current record. On output to a file connected for sequential or stream access, a new empty record is created following the current record; this new record then becomes the last and current record of the file and the file is positioned at the beginning of this new record.

3 For a file connected for direct access, the record number is increased by one and the file is positioned at the beginning of the record that has that record number, if there is such a record, and this record becomes the current record.

NOTE

A record that contains no characters can be written on output; if the file is an [internal file](#) or a file connected for direct access, the record is filled with blank characters.

An entire record can be skipped on input.

4 The repeat specification is optional in the slash edit descriptor. If it is not specified, the default value is one.

13.8.3 Colon editing

1 The colon edit descriptor terminates format control if there are no more [effective items](#) in the input/output list (12.6.3). The colon edit descriptor has no effect if there are more [effective items](#) in the input/output

list.

13.8.4 SS, SP, and S editing

1 The SS, SP, and S edit descriptors temporarily change (12.5.2) the sign mode (12.5.6.18, 12.6.2.15) for the connection. The edit descriptors SS, SP, and S set the sign mode corresponding to the SIGN= specifier values SUPPRESS, PLUS, and PROCESSOR_DEFINED, respectively.

2 The sign mode controls optional plus characters in numeric output fields. When the sign mode is PLUS, the processor shall produce a plus sign in any position that normally contains an optional plus sign. When the sign mode is SUPPRESS, the processor shall not produce a plus sign in such positions. When the sign mode is PROCESSOR_DEFINED, the processor has the option of producing a plus sign or not in such positions, subject to 13.7.2(5).

3 The SS, SP, and S edit descriptors affect only I, F, E, EN, ES, EX, D, and G editing during the execution of an output statement. The SS, SP, and S edit descriptors have no effect during the execution of an input statement.

13.8.5 LZS, Lzp and LZ editing

1 The LZS, Lzp, and LZ edit descriptors temporarily change (12.5.2) the leading zero mode for the connection. The edit descriptors LZS, Lzp, and LZ set the leading zero mode corresponding to the LEADING_ZERO= specifier (12.5.6.12, 12.6.2.10) values SUPPRESS, PRINT, and PROCESSOR_DEFINED, respectively.

2 The leading zero mode controls optional leading zero characters in numeric output fields. When the leading zero mode is PRINT, the processor shall produce a leading zero in any position that normally contains an optional leading zero. When the leading zero mode is SUPPRESS, the processor shall not produce a leading zero in such positions. When the leading zero mode is PROCESSOR_DEFINED, the processor has the option of producing a leading zero or not in such positions, subject to 13.7.2(5).

3 The LZS, Lzp, and LZ edit descriptors affect only F, E, D, and G editing during the execution of an output statement. The LZS, Lzp, and LZ edit descriptors have no effect during the execution of an input statement.

13.8.6 P editing

1 The *k*P edit descriptor temporarily changes (12.5.2) the scale factor for the connection to *k*. The scale factor affects the editing done by the F, E, EN, ES, EX, D, and G edit descriptors for real and complex quantities.

2 The scale factor *k* affects the appropriate editing in the following manner.

- On input, with F, E, EN, ES, EX, D, and G editing (provided that no exponent exists in the field), the effect is that the externally represented number equals the internally represented number multiplied by 10^k ; the scale factor is applied to the external decimal value and then this is converted using the input/output rounding mode.
- On input, with F, E, EN, ES, EX, D, and G editing, the scale factor has no effect if there is an exponent in the field.
- On output, with F output editing, the effect is that the externally represented number equals the internally represented number multiplied by 10^k ; the internal value is converted using the input/output rounding mode and then the scale factor is applied to the converted decimal value.
- On output, with E and D editing, the effect is that the significand (R715) part of the quantity to be produced is multiplied by 10^k and the exponent is reduced by *k*.

- On output, with G editing, the effect is suspended unless the magnitude of the datum to be edited is outside the range that permits the use of F editing. If the use of E editing is required, the scale factor has the same effect as with E output editing.
- On output, with EN, ES, and EX editing, the scale factor has no effect.

13.8.7 BN and BZ editing

- 1 The BN and BZ edit descriptors temporarily change (12.5.2) the blank interpretation mode for the connection. The edit descriptors BN and BZ set the blank interpretation mode corresponding to the **BLANK= specifier** (12.5.6.6, 12.6.2.6) values NULL and ZERO, respectively.
- 2 The blank interpretation mode controls the interpretation of nonleading blanks in numeric input fields. Such blank characters are interpreted as zeros when the blank interpretation mode has the value ZERO; they are ignored when the blank interpretation mode has the value NULL. The effect of ignoring blanks is to treat the input field as if blanks had been removed, the remaining portion of the field right justified, and the blanks replaced as leading blanks. However, a field containing only blanks has the value zero.
- 3 The blank interpretation mode affects only numeric editing (13.7.2) and generalized numeric editing (13.7.5.2) on input. It has no effect on output.

13.8.8 RU, RD, RZ, RN, RC, and RP editing

- 1 The round edit descriptors temporarily change (12.5.2) the connection's input/output rounding mode (12.5.6.17, 12.6.2.14, 13.7.2.3.8). The round edit descriptors RU, RD, RZ, RN, RC, and RP set the input/output rounding mode corresponding to the **ROUND= specifier** values UP, DOWN, ZERO, NEAREST, COMPATIBLE, and PROCESSOR_DEFINED, respectively. The input/output rounding mode affects the conversion of real and complex values in formatted input/output. It affects only D, E, EN, ES, EX, F, and G editing.

13.8.9 DC and DP editing

- 1 The decimal edit descriptors temporarily change (12.5.2) the decimal edit mode for the connection. The edit descriptors DC and DP set the decimal edit mode corresponding to the **DECIMAL= specifier** (12.5.6.7, 12.6.2.7, 13.6) values COMMA and POINT, respectively.
- 2 The decimal edit mode controls the representation of the **decimal symbol** (13.6) during conversion of real and complex values in formatted input/output. The decimal edit mode affects only D, E, EN, ES, EX, F, and G editing.

13.9 Character string edit descriptors

- 1 A character string edit descriptor shall not be used on input.
- 2 The character string edit descriptor causes characters to be written from the enclosed characters of the edit descriptor itself, including blanks. For a character string edit descriptor, the width of the field is the number of characters between the delimiting characters. Within the field, two consecutive delimiting characters are counted as a single character.

NOTE

A delimiter for a character string edit descriptor is either an apostrophe or quote.

13.10 List-directed formatting

13.10.1 Purpose of list-directed formatting

- 1 List-directed input/output allows data editing according to the type of the [effective item](#) instead of by a format specification. It also allows data to be free-field, that is, separated by commas (or semicolons) or blanks.

13.10.2 Values and value separators

- 1 The characters in one or more list-directed records constitute a sequence of values and value separators. The end of a record has the same effect as a blank character, unless it is within a character constant. Any sequence of two or more consecutive blanks is treated as a single blank, unless it is within a character constant.
- 2 Each value is either a null value, c , $r*c$, or r^* , where c is a literal constant, optionally signed if integer or real, or an undelimited character constant and r is an unsigned, nonzero, integer literal constant. Neither c nor r shall have kind type parameters specified. The constant c is interpreted as though it had the same kind type parameter as the corresponding [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 appearances of the null value. Neither of these forms shall 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 1

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 2

If no [effective item](#) is specified in a list-directed input/output statement, one input record is skipped or one empty output record is written.

13.10.3 List-directed input

13.10.3.1 List-directed input forms

- 1 Input forms acceptable to edit descriptors for a given type are acceptable for list-directed formatting, except as noted below. If the form of the input value is not acceptable to the processor for the type of the next [effective item](#) in the list, an error condition occurs. Blanks are never used as zeros, and embedded blanks are not permitted in constants, except within character constants and complex constants as specified below.
- 2 For the $r*c$ form of an input value, the constant c is interpreted as an undelimited character constant if the first [effective 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 1

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 or of an enum type, the value in the input record is interpreted as if an **Lw** edit descriptor with a suitable value of w were used.

4 When the next **effective item** is of type real, the input form is that of a numeric input field. A numeric input field is a field suitable for F editing (13.7.2.3.2) that is assumed to have no fractional digits unless a **decimal symbol** appears within the field.

5 When the next **effective item** is of type complex, the input form consists of a left parenthesis followed by an ordered pair of numeric input fields separated by a comma (if the decimal edit mode is POINT) or semicolon (if the decimal edit mode is COMMA), and followed by a right parenthesis. The first numeric input field is the real part of the complex constant and the second is the imaginary part. Each of the numeric input fields may be preceded or followed by any number of blanks and ends of records. The end of a record may occur after the real part or before the imaginary part.

6 When the next **effective item** is of type logical, the input form shall not include value separators among the optional characters permitted for L editing.

7 When the next **effective item** is of type character, the input form consists of a possibly delimited sequence of zero or more *rep-chars* whose kind type parameter is implied by the kind of the **effective item**. Character sequences may be continued from the end of one record to the beginning of the next record, but the end of record shall not occur between a doubled apostrophe in an apostrophe-delimited character sequence, nor between a doubled quote in a quote-delimited character sequence. The end of the record does not cause a blank or any other character to become part of the character sequence. The character sequence may be continued on as many records as needed. The characters blank, comma, semicolon, and slash may appear in default, **ASCII**, or **ISO 10646 character** sequences.

8 If the next **effective item** is default, **ASCII**, or **ISO 10646 character** and

- the character sequence does not contain value separators,
- the character sequence does not cross a record boundary,
- the first nonblank character is not a quotation mark or an apostrophe,
- the leading characters are not *digits* followed by an asterisk, and
- the character sequence contains at least one character,

the delimiting apostrophes or quotation marks are not required. If the delimiters are omitted, the character sequence is terminated by the first blank, comma (if the decimal edit mode is POINT), semicolon (if the decimal edit mode is COMMA), slash, or end of record; in this case apostrophes and quotation marks within the datum are not to be doubled.

9 Let len be the current length of the next **effective item**, and let w be the length of the character sequence. If len is less than or equal to w , the leftmost len characters of the sequence are transmitted to the next **effective item**. If len is greater than w , the sequence is transmitted to the leftmost w characters of the next **effective item** and the remaining $len - w$ characters of the next **effective item** are filled with blanks.

NOTE 2

An [allocatable](#), [deferred-length](#) character [effective item](#) does not have its allocation status or allocated length changed as a result of list-directed input.

13.10.3.2 Null values

1 A null value is specified by

- the r^* form,
- no characters between consecutive value separators, or
- no characters before the first value separator in the first record read by each execution of a list-directed [input statement](#).

NOTE 1

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 [effective items](#), 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 [effective items](#).

NOTE 2

All blanks encountered during list-directed input are considered to be part of some value separator except for

- blanks embedded in a character sequence,
- embedded blanks surrounding the real or imaginary part of a complex constant, and
- leading blanks in the first record read by each execution of a list-directed [input statement](#), unless immediately followed by a slash or comma.

NOTE 3

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:

```
12345,12345,,2*1.5,4*
ISN'T_BOB'S,(123,0),.TEXAS$
```

The results are:

NOTE 3 (cont.)

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

13.10.4 List-directed output

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 \(12.6.4.8\)](#) is determined by the [defined output](#) procedure; this form need not be compatible with list-directed input.

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.

Logical output values are T for the value true and F for the value false.

Integer output constants are produced with the effect of an [Iw](#) edit descriptor.

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 [OPFw.d](#); otherwise, [1PEw.d Ee](#) is used.

For numeric output, reasonable processor-dependent values of [w](#), [d](#), and [e](#) are used for each of the numeric constants output.

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 shall not occur between the separator and the imaginary part unless the entire constant is as long as, or longer than, an entire record. The only embedded blanks permitted within a complex constant are between the separator and the end of a record and one blank at the beginning of the next record.

Character sequences produced when the delimiter mode has a value of NONE

- are not delimited by apostrophes or quotation marks,
- are not separated from each other by value separators,

- have each internal apostrophe or quotation mark represented externally by one apostrophe or quotation mark, and
- have a blank character inserted by the processor at the beginning of any record that begins with the continuation of a character sequence from the preceding record.

Character sequences produced when the delimiter mode has a value of QUOTE are delimited by quotes, are preceded and followed by a value separator, and have each internal quote represented on the external medium by two contiguous quotes.

Character sequences produced when the delimiter mode has a value of APOSTROPHE are delimited by apostrophes, are preceded and followed by a value separator, and have each internal apostrophe represented on the external medium by two contiguous apostrophes.

If two or more successive values in an output record have identical values, the processor has the option of producing a repeated constant of the form $r*c$ instead of the sequence of identical values.

Slashes, as value separators, and null values are not produced as output by list-directed formatting.

Except for new records created by explicit formatting within a [defined output](#) procedure or by continuation of delimited character sequences, each output record begins with a blank character.

NOTE

The length of the output records is not specified and is processor dependent.

13.11 Namelist formatting

13.11.1 Purpose of namelist formatting

Namelist input/output allows data editing with name-value subsequences. This facilitates documentation of input and output files and more flexibility on input.

13.11.2 Name-value subsequences

The characters in one or more namelist records constitute a sequence of name-value subsequences, each of which consists of an [object designator](#) followed by an equals and followed by one or more values and value separators. The equals may optionally be preceded or followed by one or more contiguous blanks. The end of a record has the same effect as a blank character, unless it is within a character constant. Any sequence of two or more consecutive blanks is treated as a single blank, unless it is within a character constant.

Each object designator shall begin with a name from the [namelist-group-object-list](#) (8.9) and shall follow the syntax of [designator](#) (R901). 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 (13.10.2), or one or more contiguous blanks between a nonblank value and the following *object designator* or namelist comment (13.11.3.6).

13.11.3 Namelist input

13.11.3.1 Overall syntax

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

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.

13.11.3.2 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 (*effective items*), in the same way that formatted input/output list items are expanded (12.6.3). The number of values following the equals shall not exceed the number of *effective items*, but may be less; in the latter case, the effect is as if sufficient null values had been appended to match any remaining *effective items*. Except as noted elsewhere in this subclause, if an input value is not acceptable to the processor for the type of the corresponding *effective item*, an error condition occurs.

NOTE

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.

1 4 Successive namelist records are read by namelist input until a slash is encountered; the remainder of the
2 record is ignored.

3 5 A namelist comment may appear after any value separator except a slash (which terminates namelist in-
4 put). A namelist comment is also permitted to start in the first nonblank position of an input record except
5 within a character literal constant.

6 13.11.3.3 Namelist input values

7 1 Each value is either a null value (13.11.3.4), c , r^*c , or r^* , where c is a literal constant, optionally signed if
8 integer or real, and r is an unsigned, nonzero, integer literal constant. A kind type parameter shall not be
9 specified for c or r . The constant c is interpreted as though it had the same kind type parameter as the
10 corresponding effective item. The r^*c form is equivalent to r successive appearances of the constant c , and
11 the r^* form is equivalent to r successive null values. Neither of these forms shall contain embedded blanks,
12 except where permitted within the constant c .

13 2 The datum c (13.11) is any input value acceptable to format specifications for a given type, except for re-
14 strictions on the form of input values specified in this subclause. The form of a real or complex value is
15 dependent on the decimal edit mode in effect (13.6). The form of an input value shall be acceptable for the
16 type of the corresponding effective item. The number and forms of the input values that may follow the
17 equals in a name-value subsequence depend on the shape and type of the object represented by the name
18 in the input record. When the name in the input record is that of a scalar variable of an intrinsic type, the
19 equals shall not be followed by more than one value. Blanks are never used as zeros, and embedded blanks
20 are not permitted in constants except within character constants and complex constants as specified in this
21 subclause.

22 3 When the next effective item is of type real, the input form of the input value is that of a numeric input field.
23 A numeric input field is a field suitable for F editing (13.7.2.3.2) that is assumed to have no fractional digits
24 unless a decimal symbol appears within the field.

25 4 When the next effective item is of type complex, the input form of the input value consists of a left paren-
26 thesis followed by an ordered pair of numeric input fields separated by a comma (if the decimal edit mode
27 is POINT) or a semicolon (if the decimal edit mode is COMMA), and followed by a right parenthesis. The
28 first numeric input field is the real part of the complex constant and the second field is the imaginary part.
29 Each of the numeric input fields may be preceded or followed by any number of blanks and ends of records.
30 The end of a record may occur between the real part and the comma or semicolon, or between the comma
31 or semicolon and the imaginary part.

32 5 When the next effective item is of type logical, the input form of the input value shall not include equals or
33 value separators among the optional characters permitted for L editing (13.7.3).

34 6 When the next effective item is of type integer or of an enum type, the value in the input record is interpreted
35 as if an lw edit descriptor with a suitable value of w were used.

36 7 When the next effective item is of type character, the input form consists of a sequence of zero or more rep-
37 chars whose kind type parameter is implied by the kind of that effective item, delimited by apostrophes or
38 quotes. Such a sequence may be continued from the end of one record to the beginning of the next record,
39 but the end of record shall not occur between a doubled apostrophe in an apostrophe-delimited sequence,
40 nor between a doubled quote in a quote-delimited sequence. The end of the record does not cause a blank
41 or any other character to become part of the sequence. The sequence may be continued on as many records
42 as needed. The characters blank, comma, semicolon, and slash may appear in such character sequences.

NOTE

The delimiters in the input form for a namelist input item of type character avoid the ambiguity that could arise between undelimited character sequences and object names. The value of the `DELIM= specifier`, if any, in the `OPEN statement` for an `external file` is ignored during namelist input (12.5.6.8).

- 1 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
 2 than or equal to *w*, the leftmost *len* characters of the sequence are transmitted to the next `effective item`. If
 3 *len* is greater than *w*, the constant is transmitted to the leftmost *w* characters of the next `effective item` and
 4 the remaining *len*—*w* characters of the next `effective item` are filled with blanks. The effect is as though the
 5 sequence were assigned to the next `effective item` in an `intrinsic assignment statement` (10.2.1.3).

6 **13.11.3.4 Null values**

- 7 1 A null value is specified by
- 8 • the *r** form,
 - 9 • blanks between two consecutive nonblank value separators following an equals,
 - 10 • a value separator that is the first nonblank character following an equals, or
 - 11 • two consecutive nonblank value separators.
- 12 2 A null value has no effect on the definition status of the corresponding `effective item`. If the `effective item` is
 13 defined, it retains its previous value; if it is undefined, it remains undefined. A null value shall not be used
 14 as either the real or imaginary part of a complex constant, but a single null value may represent an entire
 15 complex constant.

NOTE

The end of a record following a value separator, with or without intervening blanks, does not specify a null value in namelist input.

16 **13.11.3.5 Blanks**

- 17 1 All blanks in a namelist input record are considered to be part of some value separator except for
- 18 • blanks embedded in a character constant,
 - 19 • embedded blanks surrounding the real or imaginary part of a complex constant,
 - 20 • leading blanks following the equals unless followed immediately by a slash or comma, or a semicolon
 21 if the decimal edit mode is COMMA, and
 - 22 • blanks between a name and the following equals.

23 **13.11.3.6 Namelist comments**

- 24 1 Except within a character literal constant, a “!” character after a value separator or in the first nonblank
 25 position of a namelist input record initiates a comment. The comment extends to the end of the record and
 26 may contain any graphic character in the processor-dependent character set. The comment is ignored. A
 27 slash within the namelist comment does not terminate execution of the namelist `input statement`. Namelist
 28 comments are not allowed in stream input because comments depend on record structure.

NOTE

Namelist input example:

NOTE (cont.)

```

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

13.11.4 Namelist output**13.11.4.1 Form of namelist output**

- The form of the output produced by intrinsic namelist output shall be suitable for input, except for character output. The names in the output are in upper case. With the exception of adjacent undelimited character values, the values are separated by one or more blanks or by a comma, or a semicolon if the decimal edit mode is COMMA, optionally preceded by one or more blanks and optionally followed by one or more blanks. The form of the output produced by [defined output \(12.6.4.8\)](#) is determined by the [defined output](#) procedure; this form need not be compatible with namelist input.

- Namelist output shall not include namelist comments.

- 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

The length of the output records is not specified exactly and is processor dependent.

13.11.4.2 Namelist output editing

- Values in namelist output records are edited as for list-directed output ([13.10.4](#)).

NOTE

Namelist output records produced with a [DELIM= specifier](#) with a value of NONE and which contain a character sequence might not be acceptable as namelist input records.

13.11.4.3 Namelist output records

- If two or more successive values for the same [namelist-group-object](#) 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.

- 1 2 The name of each *namelist-group-object* is placed in the output record followed by an equals and a list of
2 values of that *namelist-group-object*.
- 3 3 An ampersand character followed immediately by a *namelist-group-name* is placed at the start of the first
4 output record to indicate which particular group of data objects is being output. A slash is placed in the
5 output record to indicate the end of the namelist formatting.
- 6 4 A null value is not produced by namelist formatting.
- 7 5 Except for new records created by explicit formatting within a *defined output* procedure or by continuation
8 of delimited character sequences, each output record begins with a blank character.

14 Program units

14.1 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.

R1401 *main-program* **is** [[program-stmt](#)]
 [[specification-part](#)]
 [[execution-part](#)]
 [[internal-subprogram-part](#)]
 [end-program-stmt](#)

R1402 *program-stmt* **is** PROGRAM *program-name*

R1403 *end-program-stmt* **is** END [PROGRAM [*program-name*]]

C1401 The *program-name* shall not be included in an [end-program-stmt](#) unless the [main-program](#) has the optional [program-stmt](#). If included, it shall be identical to the *program-name* specified in the [program-stmt](#).

NOTE 1

The program name is global to the program (19.2). For explanatory information about uses for the program name, see C.10.1.

NOTE 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.

14.2 Modules

14.2.1 Module syntax and semantics

- 1 A [module](#) contains declarations, specifications, and definitions. Public identifiers of module entities are accessible to other [program units](#) by [use association](#) as specified in 14.2.2. A module that is provided as an inherent part of the processor is an intrinsic module. A nonintrinsic module is defined by a module [program unit](#) or a means other than Fortran.

1 2 Procedures and types defined in an intrinsic module are not themselves intrinsic.

2 R1404 *module* **is** *module-stmt*
 3 [*specification-part*]
 4 [*module-subprogram-part*]
 5 *end-module-stmt*

6 R1405 *module-stmt* **is** MODULE *module-name*

7 R1406 *end-module-stmt* **is** END [MODULE [*module-name*]]

8 R1407 *module-subprogram-part* **is** *contains-stmt*
 9 [*module-subprogram*] ...

10 R1408 *module-subprogram* **is** *function-subprogram*
 11 **or** *subroutine-subprogram*
 12 **or** *separate-module-subprogram*

13 C1402 (R1404) If the *module-name* is specified in the *end-module-stmt*, it shall be identical to the *module-*
 14 *name* specified in the *module-stmt*.

15 C1403 (R1404) A module *specification-part* shall not contain a *stmt-function-stmt*, an *entry-stmt*, or a
 16 *format-stmt*.¹

17 3 If a procedure declared in the *scoping unit* of a module has an *implicit interface*, it shall be given the **EX-**
 18 **TERNAL attribute** in that *scoping unit*; if it is a function, its type and type parameters shall be explicitly
 19 declared in a *type declaration statement* in that *scoping unit*.

20 4 If an intrinsic procedure is declared in the *scoping unit* of a module, it shall explicitly be given the **INTRINSIC**
 21 **attribute** in that *scoping unit* or be used as an intrinsic procedure in that *scoping unit*.

NOTE 1

The module name is global to the program (19.2).

NOTE 2

Although *statement function definitions*, **ENTRY statements**, and **FORMAT statements** cannot appear in the specification part of a module, they can appear in the specification part of a module subprogram in the module.

NOTE 3

For a discussion of the impact of modules on dependent compilation, see C.10.2.

NOTE 4

For examples of the use of modules, see C.10.3.

22 14.2.2 The USE statement and use association

23 1 The USE statement specifies *use association*. A USE statement is a *reference* to the module it specifies. At
 24 the time a USE statement is processed, the public portions of the specified module shall be available. A
 25 module shall not reference itself, either directly or indirectly.

¹*ENTRY statements* and *statement function statements* are obsolescent.

2 The USE statement provides the means by which a [scoping unit](#) accesses named data objects, nonintrinsic 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 an accessed 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 accessed entities are identified by the same identifiers in the [scoping unit](#) containing the USE statement, but it is possible to specify that different identifiers are used. A use-associated variable is considered to have been previously declared; any other use-associated entity is considered to have been previously defined.

NOTE 1

The accessibility of module entities can be controlled by accessibility attributes ([7.5.2.2](#), [8.5.2](#)), and the ONLY option of the USE statement. Definability of module entities can be controlled by the [PROTECTED attribute](#) ([8.5.15](#)).

12 R1409 *use-stmt* **is** USE [[, [module-nature](#)] ::] *module-name* [, [rename-list](#)]
 13 **or** USE [[, [module-nature](#)] ::] *module-name* , ■
 14 ■ ONLY : [[only-list](#)]

15 R1410 *module-nature* **is** INTRINSIC
 16 **or** NON_INTRINSIC

17 R1411 *rename* **is** *local-name* => *use-name*
 18 **or** OPERATOR ([local-defined-operator](#)) => ■
 19 ■ OPERATOR ([use-defined-operator](#))

20 R1412 *only* **is** [generic-spec](#)
 21 **or** [only-use-name](#)
 22 **or** [rename](#)

23 R1413 *only-use-name* **is** *use-name*

24 C1404 (R1409) If [module-nature](#) is INTRINSIC, *module-name* shall be the name of an intrinsic module.

25 C1405 (R1409) If [module-nature](#) is NON_INTRINSIC, *module-name* shall be the name of a nonintrinsic
 26 module.

27 C1406 (R1409) A [scoping unit](#) shall not directly reference an intrinsic module and a nonintrinsic module
 28 of the same name.

29 C1407 (R1411) OPERATOR ([use-defined-operator](#)) shall not identify a [type-bound generic interface](#).

30 C1408 (R1412) The [generic-spec](#) shall not identify a [type-bound generic interface](#).

NOTE 2

Constraints C1407 and C1408 do not prevent accessing a [generic-spec](#) that is declared by an [interface block](#), even if a [type-bound generic interface](#) has the same [generic-spec](#).

31 C1409 Each [generic-spec](#), *use-name*, and [use-defined-operator](#) in a USE statement shall be a public identi-
 32 fier of the module.

33 C1410 An [only-use-name](#) shall be a nongeneric name.

- 1 R1414 *local-defined-operator* is *defined-unary-op*
 2 or *defined-binary-op*
- 3 R1415 *use-defined-operator* is *defined-unary-op*
 4 or *defined-binary-op*
- 5 3 A *use-stmt* without a *module-nature* provides access either to an intrinsic or to a nonintrinsic module. If
 6 the *module-name* is the name of both an intrinsic and a nonintrinsic module, the nonintrinsic module is
 7 accessed.
- 8 4 The USE statement without the **ONLY** option provides access to all public entities in the specified module.
- 9 5 A USE statement with the **ONLY** option provides access only to those entities that appear as *generic-specs*,
 10 *use-names*, or *use-defined-operators* in the *only-list*.
- 11 6 More than one USE statement for a given module may appear in a specification part. If one of the USE state-
 12 ments is without an **ONLY** option, all public entities in the module are accessible. If all the USE statements
 13 have **ONLY** options, only those entities in one or more of the *only-lists* are accessible.
- 14 7 An accessible entity in the referenced module is associated with one or more accessed entities, each with
 15 its own identifier. These identifiers are
- 16 • the identifier of the entity in the referenced module if that identifier appears as an *only-use-name* or
 17 as the *defined-operator* of a *generic-spec* in any *only* for that module,
 - 18 • each of the *local-names* or *local-defined-operators* that the entity is given in any *rename* for that mod-
 19 ule, and
 - 20 • the identifier of the entity in the referenced module if that identifier does not appear as a *use-name*
 21 or *use-defined-operator* in any *rename* for that module.
- 22 8 An ultimate entity is a module entity that is not accessed by use association. An accessed entity shall not
 23 be associated with two or more ultimate entities unless its identifier is not used, or the ultimate entities
 24 are generic interfaces. **Generic interfaces** are handled as described in 15.4.3.4.

NOTE 3

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.

- 25 9 The local identifier of an entity made accessible by a USE statement shall not appear in any other nonex-
 26 ecutable statement that would cause any *attribute* (8.5) of the entity to be specified in the *scoping unit* that
 27 contains the USE statement, except that it may appear in a **PUBLIC** or **PRIVATE** statement in the *scoping*
 28 *unit* of a module and it may be given the **ASYNCHRONOUS** or **VOLATILE** attribute.
- 29 10 An entity in a *scoping unit* that is accessed by use association through more than one use path, has the
 30 **ASYNCHRONOUS** or **VOLATILE** attribute in any of those use paths, and is not given that attribute in that
 31 *scoping unit*, shall have that attribute in all use paths.

NOTE 4

The constraints in 8.10.2, 8.10.3, and 8.9 prohibit the *local-name* from appearing as a *common-block-object* in a **COMMON** statement, an *equivalence-object* in an **EQUIVALENCE** statement, or a *namelist-group-name* in a **NAMELIST** statement, respectively. There is no prohibition against the *local-name* appearing as a *namelist-group-object*. Restrictions on *local-name* being the same as a *common-block-name* are detailed in 19.3.2. The **COMMON** and **EQUIVALENCE** statements are obsolescent.

NOTE 5

For a discussion of the impact of the [ONLY](#) option and renaming on dependent compilation, see [C.10.2.2](#).

NOTE 6

Examples:

```
USE STATS_LIB
provides access to all public entities in the module STATS_LIB.
```

```
USE MATH_LIB; USE STATS_LIB, SPROD => PROD
provides access to all public identifiers in both MATH_LIB and STATS_LIB. If MATH_LIB contains an entity named PROD, it can be accessed by that name, while the entity PROD of STATS_LIB can be accessed by the name SPROD.
```

```
USE STATS_LIB, ONLY: YPROD; USE STATS_LIB, ONLY : PROD
provides access to YPROD and PROD in STAT_LIB.
```

```
USE STATS_LIB, ONLY : YPROD; USE STATS_LIB
provides access to all public identifiers in STAT_LIB.
```

14.2.3 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

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 ([15.6.2.5](#)), each of which is declared ([15.4.3.2](#)) within that submodule or one of its ancestors, and declarations and definitions of other entities that are accessible by **host association** in its **descendants**.

```
R1416 submodule          is  submodule-stmt
                           [ specification-part ]
                           [ module-subprogram-part ]
                           end-submodule-stmt
```

```
R1417 submodule-stmt     is  SUBMODULE ( parent-identifier ) submodule-name
```

```
R1418 parent-identifier  is  ancestor-module-name [ : parent-submodule-name ]
```

```
R1419 end-submodule-stmt is  END [ SUBMODULE [ submodule-name ] ]
```

C1411 A submodule **specification-part** shall not contain a **format-stmt**, **entry-stmt**, or **stmt-function-stmt**.²

C1412 (R1418) The **ancestor-module-name** shall be the name of a nonintrinsic module that declares a separate module procedure; the **parent-submodule-name** shall be the name of a **descendant** of that

²ENTRY and statement function statements are obsolescent.

1 module.

2 C1413 (R1416) If a *submodule-name* appears in the *end-submodule-stmt*, it shall be identical to the one in
3 the *submodule-stmt*.

4 14.3 Block data program units

5 1 A block data program unit is used to provide initial values for data objects in named **common blocks**.

6 2 Block data program units are obsolescent.

7 R1420 *block-data* is *block-data-stmt*
8 [*specification-part*]
9 *end-block-data-stmt*

10 R1421 *block-data-stmt* is BLOCK DATA [*block-data-name*]

11 R1422 *end-block-data-stmt* is END [BLOCK DATA [*block-data-name*]]

12 C1414 (R1420) The *block-data-name* shall be included in the *end-block-data-stmt* only if it was provided
13 in the *block-data-stmt* and, if included, shall be identical to the *block-data-name* in the *block-data-*
14 *stmt*.

15 C1415 A *block-data specification-part* shall contain only derived-type definitions and **ASYNCHRONOUS**,
16 **BIND**, **COMMON**, **DATA**, **DIMENSION**, **EQUIVALENCE**, **IMPLICIT**, **INTRINSIC**, **PARAMETER**, **POINTER**,
17 **SAVE**, **TARGET**, **USE**, **VOLATILE**, and **type declaration** statements.

18 C1416 (R1420) A **type declaration statement** in a *block-data specification-part* shall not contain **ALLOC-**
19 **ATABLE**, **EXTERNAL**, or **BIND** attribute specifiers.

20 3 If an object in a named **common block** is initially defined, all **storage units** in the common block **storage**
21 **sequence** shall be specified even if they are not all initially defined. More than one named **common block**
22 may have objects initially defined in a single block data program unit.

23 4 An object that is initially defined in a block data program unit shall be in a named **common block**.

24 5 The same named **common block** shall not be specified in more than one block data program unit in a pro-
25 gram.

26 6 There shall not be more than one unnamed block data program unit in a program.

15 Procedures

15.1 Concepts

1 The concept of a procedure was introduced in 5.2.3. This clause contains a complete description of procedures. The actions specified by a procedure are performed when the procedure is invoked by execution of a reference to it.

2 The sequence of actions encapsulated by a procedure has access to entities in the procedure reference by way of [argument association](#) (15.5.2). A name that appears as a *dummy-arg-name* in the **SUBROUTINE**, **FUNCTION**, or **ENTRY**¹ statement in the declaration of a procedure (R1539) is a **dummy argument**. **Dummy arguments** are also specified for intrinsic procedures and procedures in intrinsic modules in Clauses 16, 17, and 18.

15.2 Procedure classifications

15.2.1 Procedure classification by reference

1 The definition of a procedure specifies it to be a function or a subroutine. A reference to a function either appears explicitly as a primary within an expression, or is implied by a [defined operation](#) (10.1.6) within an expression. A reference to a subroutine is a **CALL statement**, a [defined assignment](#) statement (10.2.1.4), the appearance of an object processed by [defined input/output](#) (12.6.4.8) in an input/output list, or [finalization](#) (7.5.6).

2 A procedure is classified as [elemental](#) if it is a procedure that can be referenced [elementally](#) (15.9).

15.2.2 Procedure classification by means of definition

15.2.2.1 Intrinsic procedures

1 A procedure that is provided as an inherent part of the processor is an intrinsic procedure.

15.2.2.2 External, internal, and module procedures

1 An [external procedure](#) is a procedure that is defined by an external subprogram or by a means other than Fortran.

2 An [internal procedure](#) is a procedure that is defined by an internal subprogram. Internal subprograms may appear in the main program, in an external subprogram, or in a module subprogram. Internal subprograms shall not appear in other internal subprograms. Internal subprograms are the same as external subprograms except that the name of the [internal procedure](#) is not a global identifier, an internal subprogram shall not contain an **ENTRY statement**, and the internal subprogram has access to host entities by [host association](#).

3 A [module procedure](#) is a procedure that is defined by a module subprogram, or a specific procedure provided by an intrinsic module.

¹The [ENTRY statement](#) is obsolescent.

4 A subprogram defines a procedure for the **SUBROUTINE** or **FUNCTION** statement. If the subprogram has one or more **ENTRY statements**, it also defines a procedure for each of them.

15.2.2.3 Dummy procedures

1 A **dummy argument** that is specified to be a procedure or appears as the procedure designator in a **procedure reference** is a **dummy procedure**. A **dummy procedure** with the **POINTER attribute** is a **dummy procedure pointer**.

15.2.2.4 Procedure pointers

1 A **procedure pointer** is a procedure that has the **POINTER attribute**. A **procedure pointer** can be **pointer associated** with an **external**, **internal**, **intrinsic**, or **module** procedure.

15.2.2.5 Statement functions (obsolescent)

1 A function that is defined by a single statement is a statement function (**15.6.4**).

15.3 Characteristics

15.3.1 Characteristics of procedures

1 The **characteristics** of a procedure are the classification of the procedure as a function or subroutine, whether it is **pure**, whether it is **simple**, whether it is **elemental**, whether it has the **BIND attribute**, the **characteristics** of its **dummy arguments**, and the **characteristics** of its **function result** if it is a function.

15.3.2 Characteristics of dummy arguments

15.3.2.1 General

1 Each **dummy argument** has the **characteristic** that it is a **dummy data object**, a **dummy procedure**, or an asterisk (alternate return indicator)².

15.3.2.2 Characteristics of dummy data objects

1 The **characteristics** of a **dummy data object** are its **declared type**, its **type parameters**, its shape (unless it is **assumed-rank**), its **corank**, its **codimensions**, its intent (**8.5.10**, **8.6.9**), whether it is optional (**8.5.12**, **8.6.10**), whether it is **allocatable** (**8.5.3**), whether it has the **ASYNCHRONOUS** (**8.5.4**), **CONTIGUOUS** (**8.5.7**), **TARGET** (**8.5.18**, **8.6.15**), **VALUE** (**8.5.19**), or **VOLATILE** (**8.5.20**) attributes, whether it is **polymorphic**, and whether it is a **pointer** (**8.5.14**, **8.6.12**). If a **type parameter** of an object or a bound of an array is not a **constant expression**, the exact dependence on the entities in the expression is a characteristic. If a rank, shape, size, type, or type parameter is assumed or **deferred**, it is a characteristic.

15.3.2.3 Characteristics of dummy procedures

1 The **characteristics** of a **dummy procedure** are the explicitness of its **interface** (**15.4.2**), its **characteristics** as a procedure if the **interface** is **explicit**, whether it is a pointer, and whether it is optional (**8.5.12**, **8.6.10**).

²Alternate return indicators are obsolescent.

15.3.2.4 Characteristics of asterisk dummy arguments

- 1 A [dummy argument](#) that is an asterisk has no other characteristic.

15.3.3 Characteristics of function results

- 1 The [characteristics](#) of a function result are its [declared type](#), [type parameters](#), [rank](#), whether it is [polymorphic](#), whether it is [allocatable](#), whether it is a [pointer](#), whether it has the [CONTIGUOUS attribute](#), and whether it is a [procedure pointer](#). If a function result is an array that is not [allocatable](#) or a pointer, its shape is a characteristic. If a type parameter of a function result or a bound of a function result array is not a [constant expression](#), the exact dependence on the entities in the expression is a characteristic. If type parameters of a function result are [deferred](#), which parameters are [deferred](#) is a characteristic. Whether the length of a character function result is assumed is a characteristic.³

15.4 Procedure interface

15.4.1 Interface and abstract interface

- 1 The [interface](#) of a procedure determines the forms of reference through which it can be invoked. The procedure's interface consists of its name, [binding label](#), generic identifiers, characteristics, and the names of its dummy arguments. The [characteristics](#) and [binding label](#) of a procedure are fixed, but the remainder of the interface may differ in differing contexts, except that for a separate module procedure body ([15.6.2.5](#)), the [dummy argument](#) names and whether it has the [NON_RECURSIVE attribute](#) shall be the same as in its corresponding module procedure interface body ([15.4.3.2](#)).
- 2 An [abstract interface](#) is a set of procedure [characteristics](#) with the [dummy argument](#) names.

15.4.2 Implicit and explicit interfaces

15.4.2.1 Interfaces and scopes

- 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.⁴

NOTE

For example, the subroutine LLS of [C.10.3.4](#) has an [explicit interface](#).

15.4.2.2 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 with an [argument keyword](#) ([15.5.2](#)),

³Assumed-length character functions are obsolescent.

⁴Statement functions are obsolescent.

- (2) the procedure is used in a context that requires it to be [pure](#) (15.7),
- (3) the procedure is used in a context that requires it to be [simple](#) (15.8),
- (4) 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](#),
- (5) 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](#),
- (6) the procedure is [elemental](#), or
- (7) the procedure has the [BIND attribute](#).

15.4.3 Specification of the procedure interface

15.4.3.1 General

- 1 The interface for an [internal](#), [external](#), [module](#), or [dummy procedure](#) is specified by a [FUNCTION](#), [SUB-ROUTINE](#), or [ENTRY](#)⁵ 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

An [interface body](#) cannot be used to describe the interface of an [internal procedure](#), a [module procedure](#) that is not a separate [module procedure](#), or an intrinsic procedure because the interfaces of such procedures are already [explicit](#). However, the name of a procedure can appear in a [PROCEDURE statement](#) in an [interface block](#) (15.4.3.2).

15.4.3.2 Interface block

R1501	<i>interface-block</i>	is	<i>interface-stmt</i> [<i>interface-specification</i>] ... <i>end-interface-stmt</i>
R1502	<i>interface-specification</i>	is or	<i>interface-body</i> <i>procedure-stmt</i>
R1503	<i>interface-stmt</i>	is or	INTERFACE [<i>generic-spec</i>] ABSTRACT INTERFACE
R1504	<i>end-interface-stmt</i>	is	END INTERFACE [<i>generic-spec</i>]
R1505	<i>interface-body</i>	is	<i>function-stmt</i> [<i>specification-part</i>]

⁵The [ENTRY statement](#) is obsolescent.

- 1 *end-function-stmt*
2 **or** *subroutine-stmt*
3 [*specification-part*]
4 *end-subroutine-stmt*
- 5 R1506 *procedure-stmt* **is** [MODULE] PROCEDURE [::] *specific-procedure-list*
- 6 R1507 *specific-procedure* **is** *procedure-name*
- 7 R1508 *generic-spec* **is** *generic-name*
8 **or** OPERATOR (*defined-operator*)
9 **or** ASSIGNMENT (=)
10 **or** *defined-io-generic-spec*
- 11 R1509 *defined-io-generic-spec* **is** READ (FORMATTED)
12 **or** READ (UNFORMATTED)
13 **or** WRITE (FORMATTED)
14 **or** WRITE (UNFORMATTED)
- 15 C1501 (R1501) An *interface-block* in a subprogram shall not contain an *interface-body* for a procedure
16 defined by that subprogram.
- 17 C1502 (R1501) If the *end-interface-stmt* includes a *generic-spec*, the *interface-stmt* shall specify the same
18 *generic-spec*, except that if one *generic-spec* has a *defined-operator* that is .LT., .LE., .GT., .GE., .EQ., or
19 .NE., the other *generic-spec* may have a *defined-operator* that is the corresponding operator <, <=,
20 >, >=, ==, or /=.
- 21 C1503 (R1503) If the *interface-stmt* is ABSTRACT INTERFACE, then the *function-name* in the *function-stmt*
22 or the *subroutine-name* in the *subroutine-stmt* shall not be the same as a keyword that specifies an
23 intrinsic type.
- 24 C1504 (R1502) A *procedure-stmt* is allowed only in an interface block that has a *generic-spec*.
- 25 C1505 (R1505) An *interface-body* of a pure procedure shall specify the intents of all dummy arguments ex-
26 cept alternate return indicators⁶, dummy procedures, and arguments with the POINTER or VALUE
27 attribute.
- 28 C1506 (R1505) An *interface-body* shall not contain a *data-stmt*, *format-stmt*, *entry-stmt*, or *stmt-function-*
29 *stmt*⁷.
- 30 C1507 (R1506) If MODULE appears in a *procedure-stmt*, each *procedure-name* in that statement shall de-
31 note a module procedure.
- 32 C1508 (R1507) A *procedure-name* shall denote a nonintrinsic procedure that has an explicit interface.
- 33 C1509 (R1501) An *interface-specification* in a generic interface block shall not specify a procedure that
34 was specified previously in any accessible interface with the same generic identifier.
- 35 1 An external or module subprogram specifies a specific interface for each procedure defined in that subpro-
36 gram.
- 37 2 An interface block introduced by ABSTRACT INTERFACE is an abstract interface block. An interface body in

⁶Alternate return indicators are obsolescent.

⁷The ENTRY statement and statement functions are obsolescent.

- 1 an **abstract interface block** specifies an **abstract interface**. An interface block with a generic specification is
 2 a **generic interface block**. An interface block with neither ABSTRACT nor a generic specification is a **specific**
 3 **interface block**.
- 4 3 The name of the entity declared by an interface body is the *function-name* in the *function-stmt* or the
 5 *subroutine-name* in the *subroutine-stmt* that begins the interface body.
- 6 4 A module procedure interface body is an interface body whose initial statement contains the keyword **MOD-**
 7 **ULE**. It specifies the interface for a separate **module procedure** (15.6.2.5). A separate **module procedure** is
 8 accessible by **use association** if and only if its interface body is declared in the specification part of a module
 9 and is public. If a corresponding (15.6.2.5) separate **module procedure** is not defined, the interface may be
 10 used to specify an explicit **specific interface** but the procedure shall not be used in any other way.
- 11 5 An interface body in a generic or specific interface block specifies the **EXTERNAL attribute** and an explicit
 12 specific interface for an **external procedure**, **dummy procedure**, or **procedure pointer**. If the name of the
 13 declared procedure is that of a **dummy argument** in the subprogram containing the interface body, the
 14 procedure is a **dummy procedure**. If the procedure has the **POINTER attribute**, it is a **procedure pointer**. If
 15 it is not a **dummy procedure** or **procedure pointer**, it is an **external procedure**.
- 16 6 An interface body specifies all of the **characteristics** of the explicit specific interface or **abstract interface**.
 17 The specification part of an interface body may specify attributes or define values for data entities that do
 18 not determine **characteristics** of the procedure. Such specifications have no effect.
- 19 7 If an explicit specific interface for an **external procedure** is specified by an interface body or a **procedure**
 20 **declaration statement** (15.4.3.6), the **characteristics** shall be consistent with those specified in the proced-
 21 ure definition, except that the interface may specify a procedure that is not **pure** even if the procedure is
 22 defined to be **pure**, and the interface may specify a procedure that is not **simple** even if the procedure is
 23 defined to be **simple**. An interface for a procedure defined by an **ENTRY statement** may be specified by us-
 24 ing the entry name as the procedure name in the interface body.⁸ If an **external procedure** does not exist in
 25 the program, an interface body for it may be used to specify an explicit specific interface but the procedure
 26 shall not be used in any other way. A procedure shall not have more than one explicit specific interface in a
 27 given **scoping unit**, except that if the interface is accessed by **use association**, there may be more than one
 28 local name for the procedure. If a procedure is accessed by **use association**, each access shall be to the same
 29 procedure declaration or definition.

NOTE 1

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 2

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)
  
```

⁸The **ENTRY statement** is obsolescent.

NOTE 2 (cont.)

```
END FUNCTION EXT3
END INTERFACE
```

This interface block specifies [explicit interfaces](#) for the three [external procedures](#) EXT1, EXT2, and EXT3. Invocations of these procedures can use [argument keywords](#) (15.5.2); for example:

```
PRINT *, EXT3 (Q = P_MASK (N+1 : N+1000), P = ACTUAL_P)
```

15.4.3.3 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.

R1510 *generic-stmt* **is** **GENERIC** [, *access-spec*] :: *generic-spec* => *specific-procedure-list*

C1510 (R1510) A *specific-procedure* in a **GENERIC** statement shall not specify a procedure that was specified previously in any accessible interface with the same generic identifier.

2 If *access-spec* appears, it specifies the accessibility (8.5.2) of *generic-spec*.

15.4.3.4 Generic interfaces**15.4.3.4.1 Generic identifiers**

1 A **generic interface block** specifies a **generic interface** for each of the procedures in the interface block. The **PROCEDURE** statement lists nonintrinsic procedures with [explicit interfaces](#) 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 *generic-spec* in a **GENERIC statement** is a **generic identifier** for all of the procedures named in its *specific-procedure-list*. The rules specifying how any two procedures with the same **generic identifier** shall differ are given in 15.4.3.4.5. They ensure that any generic invocation applies to at most one specific procedure. If a specific procedure in a **generic interface** has a function **dummy argument**, that argument shall have its type and type parameters explicitly declared in the **specific interface**.

3 A generic name is a **generic identifier** that refers to all of the procedure names in the **generic interface**. A generic name may be the same as any one of the procedure names in the **generic interface**, 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 **generic interface** shall be functions.

5 An *interface-stmt* having a *defined-io-generic-spec* is an interface for a **defined input/output** procedure (12.6.4.8).

NOTE 1

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 1 (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 2

A [type-bound-generic-stmt](#) within a derived-type definition (7.5.5) specifies a [generic identifier](#) for a set of [type-bound procedures](#).

15.4.3.4.2 Defined operations

- 1 If [OPERATOR](#) is specified in a generic specification, all of the procedures specified in the [generic interface](#) shall be functions that can be referenced as [defined operations](#) (10.1.6, 15.5). In the case of functions of two arguments, infix binary operator notation is implied. In the case of functions of one argument, prefix operator notation is implied. [OPERATOR](#) shall not be specified for functions with no arguments or for functions with more than two arguments. The [dummy arguments](#) shall be nonoptional [dummy data objects](#) and shall have the [INTENT \(IN\)](#) or [VALUE](#) attribute. The function result shall not have assumed character length⁹. If the operator is an *intrinsic-operator* (R608), the number of [dummy arguments](#) shall be consistent with the intrinsic uses of that operator, and the types, kind type parameters, or [ranks](#) of the [dummy arguments](#) shall differ from those required for the intrinsic operation (10.1.5), treating a CLASS (*) [dummy argument](#) as not differing in type or kind.
- 2 A [defined operation](#) is treated as a reference to the function. For a unary [defined operation](#), the operand corresponds to the function's [dummy argument](#); for a binary operation, the left-hand operand corresponds to the first [dummy argument](#) of the function and the right-hand operand corresponds to the second [dummy argument](#). All restrictions and constraints that apply to [actual arguments](#) in a [reference](#) to the function also apply to the corresponding operands in the expression as if they were used as [actual arguments](#).
- 3 A given defined operator may, as with generic names, apply to more than one function, in which case it is generic in exact analogy to generic procedure names. For intrinsic operator symbols, the generic properties include the intrinsic operations they represent. Because both forms of each relational operator have the same interpretation (10.1.6.2), extending one form (such as <=) has the effect of defining both forms (<= and [.LE.](#)).

NOTE

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 reference

⁹Assumed character length functions are obsolescent.

NOTE (cont.)

BOOLEAN_AND (SENSOR (1:N), ACTION (1:N)) ! SENSOR and ACTION are of type LOGICAL

15.4.3.4.3 Defined assignments

- 1 If **ASSIGNMENT** (=) is specified in a generic specification, all the procedures in the **generic interface** shall be subroutines that can be referenced as **defined assignments** (10.2.1.4, 10.2.1.5). **Defined assignment** may, as with generic names, apply to more than one subroutine, in which case it is generic in exact analogy to generic procedure names.
- 2 Each of these subroutines shall have exactly two **dummy arguments**. The **dummy arguments** shall be non-optional **dummy data objects**. The first argument shall have **INTENT (OUT)** or **INTENT (INOUT)** and the second argument shall have the **INTENT (IN)** or **VALUE** attribute. Either the second argument shall be an array whose **rank** differs from that of the first argument, the **declared types** and **kind type parameters** of the arguments shall not conform as specified in Table 10.8, or the first argument shall be of derived type. A **defined assignment** is treated as a reference to the subroutine, with the left-hand side as the first argument and the right-hand side enclosed in parentheses as the second argument. All restrictions and constraints that apply to **actual arguments** in a reference to the subroutine also apply to the left-hand-side and to the right-hand-side enclosed in parentheses as if they were used as **actual arguments**. The **ASSIGNMENT** generic specification specifies that assignment is extended or redefined.

NOTE 1

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 2

A procedure which has a generic identifier of **ASSIGNMENT** (=) and whose second **dummy argument** has the **ALLOCATABLE** or **POINTER** attribute cannot be directly invoked by **defined assignment**. This is because the **actual argument** associated with that **dummy argument** is the right-hand side of the assignment enclosed in parentheses, which makes the **actual argument** an expression that does not have the **ALLOCATABLE**, **POINTER**, or **TARGET** attribute.

15.4.3.4.4 Defined input/output procedure interfaces

- 1 All of the procedures specified in an interface block for a **defined input/output** procedure shall be subroutines that have interfaces as described in 12.6.4.8.2.

15.4.3.4.5 Restrictions on generic declarations

- 1 This subclause contains the rules that shall be satisfied by every pair of specific procedures that have the same **generic identifier** within the scope of the identifier. If a generic procedure is accessed from a module, the rules apply to all the specific versions even if some of them are inaccessible by their **specific names**.

NOTE 1

In most **scoping units**, the possible sources of procedures with a particular **generic identifier** are the accessible generic identifiers specified by **generic interface blocks** or **GENERIC statements** and the generic **bindings** other than names for the accessible objects in that **scoping unit**. In a type definition, they are the generic **bindings**, including those from a **parent type**.

- 2 A **dummy argument** is type, kind, and **rank** compatible, or TKR compatible, with another **dummy argument** if the first is **type compatible** with the second, the kind type parameters of the first have the same values as the corresponding kind type parameters of the second, and both have the same **rank** or either is **assumed-rank**.
- 3 Two **dummy arguments** are distinguishable if
- one is a procedure and the other is a data object,
 - they are both data objects or known to be functions, and neither is TKR compatible with the other,
 - one has the **ALLOCATABLE attribute** and the other has the **POINTER attribute** and not the **INTENT (IN) attribute**, or
 - one is a function with nonzero **rank** and the other is not known to be a function.
- C1511 Within the scope of a generic operator, if two procedures with that identifier have the same number of arguments, one shall have a **dummy argument** that corresponds by position in the argument list to a **dummy argument** of the other that is distinguishable from it.
- C1512 Within the scope of the generic **ASSIGNMENT (=)** identifier, if two procedures have that identifier, one shall have a **dummy argument** that corresponds by position in the argument list to a **dummy argument** of the other that is distinguishable from it.
- C1513 Within the scope of a *defined-io-generic-spec*, if two procedures have that **generic identifier**, their **dtv arguments (12.6.4.8.2)** shall be distinguishable.
- C1514 Within the scope of a generic name, each pair of procedures identified by that name shall both be subroutines or both be functions, and
- (1) there is a non-passed-object **dummy data object** in one or the other of them such that
 - (a) the number of **dummy data objects** in one that are nonoptional, are not passed-object, and with which that **dummy data object** is TKR compatible, possibly including that **dummy data object** itself, exceeds
 - (b) the number of non-passed-object **dummy data objects**, both optional and nonoptional, in the other that are not distinguishable from that **dummy data object**,
 - (2) the number of nonoptional **dummy procedures** in one of them exceeds the number of **dummy procedures** in the other,
 - (3) both have **passed-object dummy arguments** and the **passed-object dummy arguments** are distinguishable, or
 - (4) at least one of them shall have both

- (a) a nonoptional non-passed-object [dummy argument](#) at an effective position such that either the other procedure has no [dummy argument](#) at that effective position or the [dummy argument](#) at that position is distinguishable from it, and
- (b) a nonoptional non-passed-object [dummy argument](#) whose name is such that either the other procedure has no [dummy argument](#) with that name or the [dummy argument](#) with that name is distinguishable from it,
- and the [dummy argument](#) that disambiguates by position shall either be the same as or occur earlier in the argument list than the one that disambiguates by name.

4 The effective position of a [dummy argument](#) is its position in the argument list after any [passed-object dummy argument](#) has been removed.

5 Within the scope of a generic name that is the same as the generic name of an intrinsic procedure, the intrinsic procedure is not accessible by its generic name if the procedures in the interface and the intrinsic procedure are not all functions or not all subroutines. If a generic invocation is consistent with both a specific procedure from an interface and an accessible intrinsic procedure, it is the specific procedure from the interface that is referenced.

NOTE 2

An extensive explanation of the application of these rules is in [C.11.6](#).

15.4.3.5 EXTERNAL statement

1 An EXTERNAL statement specifies the [EXTERNAL attribute](#) (8.5.9) for a list of names.

R1511 *external-stmt* **is** EXTERNAL [*::*] *external-name-list*

2 The appearance of the name of a [block data program unit](#) in an EXTERNAL statement confirms that the [block data program unit](#) is a part of the program.

NOTE 1

For explanatory information on potential portability problems with [external procedures](#), see [C.11.1](#).

NOTE 2

An example of an EXTERNAL statement is:

EXTERNAL FOCUS

15.4.3.6 Procedure declaration statement

1 A procedure declaration statement declares [procedure pointers](#), [dummy procedures](#), and [external procedures](#). It specifies the [EXTERNAL attribute](#) (8.5.9) for all entities in the *proc-decl-list*.

R1512 *procedure-declaration-stmt* **is** PROCEDURE ([*proc-interface*]) ■
 ■ [[, *proc-attr-spec*] ... *::*] *proc-decl-list*

R1513 *proc-interface* **is** *interface-name*
 or *declaration-type-spec*

R1514 *proc-attr-spec* **is** *access-spec*
 or *proc-language-binding-spec*

or INTENT (*intent-spec*)
or OPTIONAL
or POINTER
or PROTECTED
or SAVE

R1515 *proc-decl* **is** *procedure-entity-name* [= > *proc-pointer-init*]

R1516 *interface-name* **is** *name*

R1517 *proc-pointer-init* **is** *null-init*
or *initial-proc-target*

R1518 *initial-proc-target* **is** *procedure-name*

C1515 (R1516) The *name* shall be the name of an **abstract interface** or of a procedure that has an **explicit interface**. If *name* is declared by a *procedure-declaration-stmt* it shall be previously declared. If *name* denotes an intrinsic procedure it shall be one that is listed in Table 16.2.¹⁰

C1516 (R1516) The *name* shall not be the same as a keyword that specifies an intrinsic type.

C1517 (R1512) If a *proc-interface* describes an **elemental procedure**, each *procedure-entity-name* shall specify an **external procedure**.

C1518 (R1515) If => appears in *proc-decl*, the procedure entity shall have the **POINTER attribute**.

C1519 (R1518) The *procedure-name* shall be the name of a nonelemental **external** or **module procedure**, or a specific intrinsic function listed in Table 16.2¹¹.

C1520 (R1512) If *proc-language-binding-spec* with **NAME=** is specified, then *proc-decl-list* shall contain exactly one *proc-decl*, which shall neither have the **POINTER attribute** nor be a **dummy procedure**.

C1521 (R1512) If *proc-language-binding-spec* is specified, the *proc-interface* shall appear, it shall be an *interface-name*, and *interface-name* shall be declared with a *proc-language-binding-spec*.

2 If *proc-interface* appears and consists of *interface-name*, it specifies an explicit **specific interface** (15.4.3.2) for the declared procedure entities. The **abstract interface** (15.4) is that specified by the interface named by *interface-name*. The interface specified by *interface-name* shall not depend on any characteristic of a procedure identified by a *procedure-entity-name* in the *proc-decl-list* of the same procedure declaration statement.

3 If *proc-interface* appears and consists of *declaration-type-spec*, it specifies that the declared procedure entities are functions having **implicit interfaces** and the specified result type. If a type is specified for an external function, its function definition (15.6.2.2) shall specify the same result type and type parameters.

4 If *proc-interface* does not appear, the procedure declaration statement does not specify whether the declared procedure entities are subroutines or functions.

5 If a *proc-attr-spec* other than a *proc-language-binding-spec* appears, it specifies that the declared procedure entities have that attribute. These attributes are described in 8.5. If a *proc-language-binding-spec* with **NAME=** appears, it specifies a **binding label** or its absence, as described in 18.10.2. A *proc-language-binding-spec* without **NAME=** is allowed, but is redundant with the *proc-interface* required by C1521.

¹⁰Specific intrinsic procedure names are obsolescent.

¹¹Specific intrinsic function names are obsolescent.

- 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*, *initial-proc-target* may be *simple* even if *procedure-entity-name* is not *simple*, and *initial-proc-target* may be an *elemental intrinsic procedure*¹².
- 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

The following code illustrates procedure declaration statements. 10.2.2.5, NOTE 1 illustrates the use of the P and BESSEL defined by this code.

```

ABSTRACT INTERFACE
  FUNCTION REAL_FUNC (X)
    REAL, INTENT (IN) :: X
    REAL :: REAL_FUNC
  END FUNCTION REAL_FUNC
END INTERFACE

INTERFACE
  SUBROUTINE SUB (X)
    REAL, INTENT (IN) :: X
  END SUBROUTINE SUB
END INTERFACE

!-- Some external or dummy procedures with explicit interface.
PROCEDURE (REAL_FUNC) :: BESSEL, GFUN
PROCEDURE (SUB) :: PRINT_REAL
!-- Some procedure pointers with explicit interface,
!-- one initialized to NULL().
PROCEDURE (REAL_FUNC), POINTER :: P, R => NULL ()
PROCEDURE (REAL_FUNC), POINTER :: PTR_TO_GFUN
!-- A derived type with a procedure pointer component ...
TYPE STRUCT_TYPE
  PROCEDURE (REAL_FUNC), POINTER, NOPASS :: COMPONENT
END TYPE STRUCT_TYPE
!-- ... and a variable of that type.
TYPE (STRUCT_TYPE) :: STRUCT
!-- An external or dummy function with implicit interface
PROCEDURE (REAL) :: PSI

```

15.4.3.7 INTRINSIC statement

- 1 An INTRINSIC statement specifies the *INTRINSIC attribute* (8.5.11) for a list of names.
- R1519 *intrinsic-stmt* **is** INTRINSIC [::] *intrinsic-procedure-name-list*
- C1522 (R1519) Each *intrinsic-procedure-name* shall be the name of an intrinsic procedure.

¹²Specific intrinsic procedure names are obsolescent.

15.4.3.8 Implicit interface specification

1 If the interface of a function is **implicit**, the type and type parameters of the function result are specified by an implicit or explicit type specification of the function name. The type, type parameters, and shape of the **dummy arguments** of a procedure invoked from where the interface of the procedure is **implicit** shall be such that each **actual argument** is consistent with the **characteristics** of the corresponding **dummy argument**.

15.5 Procedure reference

15.5.1 Syntax of a procedure reference

1 The form of a procedure reference is dependent on the **interface** of the procedure or **procedure pointer**, but is independent of the means by which the procedure is defined. The forms of procedure references are as follows.

R1520 *function-reference* **is** *procedure-designator* ([*actual-arg-spec-list*])

C1523 (R1520) The *procedure-designator* shall designate a function.

C1524 (R1520) The *actual-arg-spec-list* shall not contain an *alt-return-spec*.

R1521 *call-stmt* **is** CALL *procedure-designator* [([*actual-arg-spec-list*])]

C1525 (R1521) The *procedure-designator* shall designate a subroutine.

R1522 *procedure-designator* **is** *procedure-name*
 or *proc-component-ref*
 or *data-ref* % *binding-name*

C1526 (R1522) A *procedure-name* shall be a generic name or the name of a procedure.

C1527 (R1522) A *binding-name* shall be a **binding name** (7.5.5) of the **declared type** of *data-ref*.

C1528 (R1522) A *data-ref* shall not be a **polymorphic** subobject of a **coindexed object**.

C1529 (R1522) If *data-ref* is an array, the referenced **type-bound procedure** shall have the **PASS attribute**.

2 The *data-ref* in a *procedure-designator* shall not be an unallocated **allocatable** variable or a pointer that is not associated.

3 Resolving references to **type-bound procedures** is described in 15.5.6.

4 A function may also be referenced as a **defined operation** (10.1.6). A subroutine may also be referenced as a **defined assignment** (10.2.1.4, 10.2.1.5), by **defined input/output** (12.6.4.8), or by **finalization** (7.5.6).

NOTE 1

When resolving **type-bound procedure** references, constraints on the use of **coindexed objects** ensure that the **coindexed object** (on the remote **image**) has the same **dynamic type** as the corresponding object on the local **image**. Thus a processor can resolve the **type-bound procedure** using the **coarray** variable on its own **image** and pass the **coindexed object** as the **actual argument**.

R1523 *actual-arg-spec* **is** [*keyword* =] *actual-arg*

R1524 *actual-arg* **is** *expr*

or *variable*
 or *procedure-name*
 or *proc-component-ref*
 or *conditional-arg*
 or *alt-return-spec*

R1525 *alt-return-spec*

is * *label*

C1530 (R1523) The *keyword* = shall not appear if the *interface* of the procedure is *implicit*.

C1531 (R1523) The *keyword* = shall not be omitted from an *actual-arg-spec* unless it has been omitted from each preceding *actual-arg-spec* in the argument list.

C1532 (R1523) Each *keyword* shall be the name of a *dummy argument* in the *explicit interface* of the procedure.

C1533 (R1524) A nonintrinsic *elemental procedure* shall not be used as an *actual argument*.

C1534 (R1524) A *procedure-name* shall be the name of an *external*, *internal*, *module*, or *dummy* procedure, a specific intrinsic function listed in Table 16.2, or a *procedure pointer*.

C1535 An *actual-arg* that is an *expr* shall not be a variable or a *conditional-arg*.

C1536 (R1525) The *label* shall be the *statement label* of a *branch target statement* that appears in the same *inclusive scope* as the *call-stmt*.

C1537 An actual argument that is a *coindexed object* shall not have a pointer *ultimate component*.

R1526 *conditional-arg*

is (*scalar-logical-expr* ? *consequent* ■
 ■ [: *scalar-logical-expr* ? *consequent*]... : *consequent*)

R1527 *consequent*

is *consequent-arg*

or .NIL.

R1528 *consequent-arg*

is *expr*

or *variable*

C1538 Each *consequent-arg* of a *conditional-arg* shall have the same *declared type*, and *kind type parameters*.

C1539 Either all *consequent-args* in a *conditional-arg* shall have the same rank, or be *assumed-rank*.

C1540 At least one *consequent* in a *conditional-arg* shall be a *consequent-arg*. If the corresponding *dummy argument* is not optional, .NIL. shall not appear.

C1541 If its corresponding *dummy argument* is INTENT (OUT) or INTENT (INOUT), each *consequent-arg* in a *conditional-arg* shall be a variable.

C1542 If its corresponding *dummy argument* is *allocatable*, a pointer, or a *coarray*, the attributes of each *consequent-arg* in a *conditional-arg* shall satisfy the requirements of that *dummy argument*.

C1543 A *consequent-arg* shall not be *assumed-rank* unless its corresponding *dummy argument* is also *assumed-rank*.

C1544 A *consequent-arg* that is an *expr* shall not be a variable.

- 1 C1545 In a reference to a generic procedure, each *consequent-arg* in a *conditional-arg* shall have the same
 2 *corank*, and if any *consequent-arg* of a *conditional-arg* has the *ALLOCATABLE* or *POINTER* attribute,
 3 each *consequent-arg* shall have that attribute.

NOTE 2

Examples of procedure reference using *procedure pointers*:

```
P => BESSEL
WRITE (*, *) P(2.5)      !-- BESSEL(2.5)

S => PRINT_REAL
CALL S(3.14)
```

NOTE 3

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 needs to be explicit and available.

CONTAINS
  REAL(C_FLOAT) FUNCTION MY_F(X) BIND(C) ! Integrand
    REAL(C_FLOAT), VALUE :: X
    MY_F = X**N ! N is taken from the host instance of MY_INTEGRATION.
  END FUNCTION
END FUNCTION MY_INTEGRATION
```

The function INTEGRATE cannot retain a function pointer to MY_F and use it after INTEGRATE has finished execution, because the *host instance* of MY_F might no longer exist, making the pointer undefined. If such a pointer is retained, then it can only be used to invoke MY_F during the execution of the instance of MY_INTEGRATION that called INTEGRATE.

15.5.2 Actual arguments, dummy arguments, and argument association

15.5.2.1 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 can be established either by keyword or by position. If an **argument keyword** appears, the **actual argument** corresponds to the **dummy argument** whose name is the same as the **argument keyword** (using the **dummy argument** names from the **interface** accessible by the **procedure reference**). In the absence of an **argument keyword**, an **actual argument** corresponds to the **dummy argument** occupying the corresponding position in the reduced **dummy argument** list; that is, the first **actual argument** corresponds to the first **dummy argument** in the reduced list, the second **actual argument** corresponds to the second **dummy argument** in the reduced list, etc. The reduced **dummy argument** list is either the full **dummy argument** list or, if there is a **passed-object dummy argument** (7.5.4.5), the **dummy argument** list with the **passed-object dummy argument** omitted. Exactly one **actual argument** shall correspond to each nonoptional **dummy argument**. At most one **actual argument** shall correspond to each optional **dummy argument**. Each **actual argument** shall correspond to a **dummy argument**.

NOTE

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 body**, the name of the last argument is PRINT.

15.5.2.2 The passed-object dummy argument and argument correspondence

1 In a reference to a **type-bound procedure**, or a **procedure pointer** component, that has a **passed-object dummy argument** (7.5.4.5), the **data-ref** of the **function-reference** or **call-stmt** corresponds, as an **actual argument**, with the **passed-object dummy argument**.

15.5.2.3 Conditional argument correspondence

1 If an **actual-arg** is a **conditional-arg**, each **scalar-logical-expr** is evaluated in order, until either the value of a **scalar-logical-expr** is true, or there are no more **scalar-logical-exprs**. If the value of a **scalar-logical-expr** is true, its subsequent **consequent** is chosen; otherwise, the last **consequent** is chosen.

2 If the chosen **consequent** is a **consequent-arg**, its **expr** or **variable** is the **actual argument** for the corresponding **dummy argument**, and if it is an **expr**, it is evaluated. If the chosen **consequent** is **.NIL.**, the **actual argument** for that **dummy argument** is not present.

3 Each **consequent-arg** in a **conditional-arg** shall satisfy any requirements of the **dummy argument** on **declared type**, **kind type parameters**, attributes, and properties that do not depend on evaluation of the **consequent-arg** or any contained expressions.

- 1 4 The **declared type**, **kind type parameters**, and **rank** of a **conditional-arg** are those of its **consequent-args**. It
 2 has the **ALLOCATABLE** or **POINTER** attribute if and only if all of its **consequent-args** have that attribute. It
 3 is **polymorphic** if and only if one or more of its **consequent-args** is **polymorphic**. If all of its **consequent-args**
 4 have the same **corank**, it has that **corank**; otherwise it has **corank** zero. It is **simply contiguous** if and only if
 5 all of its **consequent-args** are **simply contiguous**.

NOTE

An example of conditional arguments in a procedure reference is:

```
CALL sub ( ( x>0 ? x : y>0 ? y : z ), &
           ( edge>0 ? edge : mode==3 ? 1.0 : .NIL. ), &
           some, other, arguments)
```

6 15.5.2.4 Argument association

- 7 1 Except in references to intrinsic **inquiry functions**, a pointer **actual argument** that corresponds to a nonop-
 8 tional nonpointer **dummy argument** shall be **pointer associated** with a **target**.
- 9 2 If a nonpointer **dummy argument** without the **VALUE attribute** corresponds to a pointer **actual argument**
 10 that is **pointer associated** with a **target**,
- 11 • if the **dummy argument** is **polymorphic**, it becomes argument associated with that **target**;
 - 12 • if the **dummy argument** is nonpolymorphic, it becomes argument associated with the **declared type**
 13 part of that **target**.
- 14 3 If a present nonpointer **dummy argument** without the **VALUE attribute** corresponds to a nonpointer **actual**
 15 **argument**,
- 16 • if the **dummy argument** is **polymorphic**, it becomes argument associated with that **actual argument**;
 - 17 • if the **dummy argument** is nonpolymorphic, it becomes argument associated with the **declared type**
 18 part of that **actual argument**.
- 19 4 A present **dummy argument** with the **VALUE attribute** becomes argument associated with a **definable** an-
 20 onymous data object whose initial value is the value of the **actual argument**.
- 21 5 A present pointer **dummy argument** that corresponds to a pointer **actual argument** becomes argument
 22 associated with that **actual argument**. A present pointer **dummy argument** that does not correspond to a
 23 pointer **actual argument** is not argument associated.
- 24 6 The entity that is argument associated with a **dummy argument** is called its **effective argument**.
- 25 7 The **ultimate argument** is the **effective argument** if the **effective argument** is not a **dummy argument** or a
 26 subobject of a **dummy argument**. If the **effective argument** is a dummy argument, the **ultimate argument**
 27 is the **ultimate argument** of that dummy argument. If the **effective argument** is a subobject of a dummy
 28 argument, the **ultimate argument** is the corresponding subobject of the **ultimate argument** of that dummy
 29 argument.

NOTE 1

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))
```


NOTE 1 (cont.)

```
...
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 2

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** were assigned to a temporary variable, and that variable were then argument associated with the **dummy argument**. Subsequent changes to the value or definition status of the dummy argument do not affect the **actual argument**. The actual mechanism by which this happens is determined by the processor.

15.5.2.5 Ordinary dummy variables

- 1 The requirements in this subclause apply to **actual arguments** that correspond to nonallocatable non-pointer dummy data objects.
- 2 The **dummy argument** shall be **type compatible** with the **actual argument**. If the **actual argument** is a **polymorphic coindexed object**, the dummy argument shall not be **polymorphic**. If the **actual argument** is a **polymorphic assumed-size array**, the **dummy argument** shall be **polymorphic**. If the **actual argument** is of a derived type that has **type parameters**, **type-bound procedures**, or **final subroutines**, the **dummy argument** shall not be **assumed-type**.
- 3 The **kind type parameter** values of the **actual argument** shall agree with the corresponding ones of the **dummy argument**. The **length type parameter** values of a present **actual argument** shall agree with the corresponding ones of the **dummy argument** that are not assumed, except for the case of the character length parameter of an **actual argument** of type character with default kind or C character kind (18.2.2) associated with a **dummy argument** that is not **assumed-shape** or **assumed-rank**.
- 4 If a present scalar **dummy argument** is of type character with default kind or C character kind, the length *len* of the **dummy argument** shall be less than or equal to the length of the **actual argument**. The **dummy argument** becomes associated with the leftmost *len* characters of the **actual argument**. If a present array **dummy argument** is of type character with default kind or C character kind and is not **assumed-shape** or **assumed-rank**, it becomes associated with the leftmost characters of the **actual argument** element sequence (15.5.2.12).
- 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 1

If the **actual argument** is a **coindexed object**, a processor that uses distributed memory might create a copy on the executing **image** of the **actual argument**, including copies of any allocated **allocatable subobjects**, and associate the **dummy argument** with that copy. If necessary, on return from the procedure, the value of the copy would be copied back to the **actual argument**.

- 7 Except in references to intrinsic **inquiry functions**, if the dummy argument is nonoptional and the **actual argument** is **allocatable**, the corresponding **actual argument** shall be allocated.
- 8 If the dummy argument does not have the **TARGET attribute**, any pointers associated with the **effective**

argument do not become associated with the corresponding dummy argument on invocation of the procedure. If such a dummy argument is used as an **actual argument** that corresponds to a dummy argument with the **TARGET attribute**, whether any pointers associated with the original **effective argument** become associated with the dummy argument with the **TARGET attribute** is processor dependent.

9 If the dummy argument has the **TARGET attribute**, does not have the **VALUE attribute**, and either the **effective argument** is **simply contiguous** or the **dummy argument** is scalar, **assumed-rank**, or **assumed-shape**, and does not have the **CONTIGUOUS attribute**, and the **effective argument** has the **TARGET attribute** but is not a **coindexed object** or an **array section** with a **vector subscript** then

- any pointers associated with the **effective argument** become associated with the corresponding dummy argument on invocation of the procedure, and
- when execution of the procedure completes, any pointers that do not become undefined (19.5.2.5) and are associated with the dummy argument remain associated with the **effective argument**.

10 If the dummy argument has the **TARGET attribute** and is an **explicit-shape array**, an **assumed-shape array** with the **CONTIGUOUS attribute**, an **assumed-rank object** with the **CONTIGUOUS attribute**, or an **assumed-size array**, and the **effective argument** has the **TARGET attribute** but is not **simply contiguous** and is not an **array section** with a **vector subscript** then

- on invocation of the procedure, whether any pointers associated with the **effective argument** become associated with the corresponding dummy argument is processor dependent, and
- when execution of the procedure completes, the **pointer association** status of any pointer that is **pointer associated** with the **dummy argument** is processor dependent.

11 If the dummy argument has the **TARGET attribute** and the **effective argument** does not have the **TARGET attribute** or is an **array section** with a **vector subscript**, any pointers associated with the dummy argument become undefined when execution of the procedure completes.

12 If the **dummy argument** has the **TARGET attribute** and the **VALUE attribute**, any pointers associated with the **dummy argument** become undefined when execution of the procedure completes.

13 If the **actual argument** is a **coindexed** scalar, the corresponding **dummy argument** shall be scalar.

14 If the **actual argument** is a noncoindexed scalar, the corresponding **dummy argument** shall be scalar unless

- the **actual argument** is default character, of type character with the C character kind (18.2.2), or is an element or substring of an element of an array that is not an **assumed-shape**, **pointer**, or **polymorphic array**,
- the **dummy argument** has **assumed-rank**, or
- the **dummy argument** is an **assumed-type assumed-size array**.

15 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.

16 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, and the **actual argument** shall not be an **assumed-size array**.

17 An **actual argument** of any **rank** may correspond to an **assumed-rank** dummy argument. The **rank** and **extents** of the dummy argument are the **rank** and **extents** of the corresponding actual argument. The lower bound of each dimension of the dummy argument is equal to one. The upper bound is equal to the **extent**, except for the last dimension when the actual argument is **assumed-size**.

- 1 18 Except when a procedure reference is [elemental](#) (15.9), each element of an array [actual argument](#) or of a
 2 sequence in a sequence association (15.5.2.12) is associated with the element of the dummy array that has
 3 the same position in array element order (9.5.3.3).

NOTE 2

For default character sequence associations, the interpretation of element is provided in [15.5.2.12](#).

- 4 19 A scalar dummy argument of a non[elemental](#) procedure shall correspond only to a scalar actual argument.
- 5 20 If a dummy argument has [INTENT \(OUT\)](#) or [INTENT \(INOUT\)](#), the [actual argument](#) shall be [definable](#). If a
 6 dummy argument has [INTENT \(OUT\)](#), the [effective argument](#) becomes undefined at the time the association
 7 is established, except for [direct components](#) of an object of derived type for which [default initialization](#) has
 8 been specified.
- 9 21 If the procedure is nonelemental, the [dummy argument](#) does not have the [VALUE attribute](#), and the [actual](#)
 10 [argument](#) is an [array section](#) having a [vector subscript](#), the dummy argument is not [definable](#) and shall not
 11 have the [ASYNCHRONOUS](#), [INTENT \(OUT\)](#), [INTENT \(INOUT\)](#), or [VOLATILE](#) attributes.
- 12 22 If the [dummy argument](#) has a [coarray potential subobject component](#), the corresponding [actual argument](#)
 13 shall have the [VOLATILE attribute](#) if and only if the [dummy argument](#) has the [VOLATILE attribute](#). If the
 14 [dummy argument](#) is an array with a [coarray potential subobject component](#), the corresponding [actual ar-](#)
 15 [gument](#) shall be [simply contiguous](#) or an element of a [simply contiguous](#) array.

NOTE 3

Argument intent specifications serve several purposes. See [8.5.10](#), NOTE 4.

NOTE 4

For more explanatory information on [targets](#) as dummy arguments, see [C.11.4](#).

- 16 C1546 An [actual argument](#) that is a [coindexed object](#) with the [ASYNCHRONOUS](#) or [VOLATILE](#) attribute
 17 shall not correspond to a [dummy argument](#) that has the [ASYNCHRONOUS attribute](#), unless the
 18 [dummy argument](#) has the [VALUE attribute](#).
- 19 C1547 An [actual argument](#) that is a [coindexed object](#) with the [ASYNCHRONOUS](#) or [VOLATILE](#) attribute
 20 shall not correspond to a [dummy argument](#) that has the [VOLATILE attribute](#).
- 21 C1548 (R1524) If an [actual argument](#) is a nonpointer array that has the [ASYNCHRONOUS](#) or [VOLATILE](#)
 22 attribute but is not [simply contiguous](#) (9.5.4), and the corresponding [dummy argument](#) has either
 23 the [ASYNCHRONOUS](#) or [VOLATILE](#) attribute, but does not have the [VALUE attribute](#), that [dummy](#)
 24 [argument](#) shall be [assumed-shape](#) or [assumed-rank](#) and shall not have the [CONTIGUOUS attribute](#).
- 25 C1549 (R1524) If an [actual argument](#) is an array pointer that has the [ASYNCHRONOUS](#) or [VOLATILE](#) at-
 26 tribute but does not have the [CONTIGUOUS attribute](#), and the corresponding [dummy argument](#) has
 27 either the [ASYNCHRONOUS](#) or [VOLATILE](#) attribute, but does not have the [VALUE attribute](#), that
 28 [dummy argument](#) shall be an array pointer, an [assumed-shape array](#) without the [CONTIGUOUS at-](#)
 29 [tribute](#), or an [assumed-rank entity](#) without the [CONTIGUOUS attribute](#).

NOTE 5

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

NOTE 5 (cont.)

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**.

NOTE 6

If an **effective argument** is a discontinuous array, and the **dummy argument** is an **assumed-shape array** with the **CONTIGUOUS** attribute, an **assumed-rank dummy data object** with the **CONTIGUOUS** attribute, an **explicit-shape array**, or an **assumed-size array**, the processor might need to use the so-called copy-in/copy-out argument passing mechanism, so as to ensure that the **dummy array** is contiguous even when the **actual argument** is not.

15.5.2.6 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

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 non-polymorphic.

- 3 The **rank** of the **actual argument** shall be the same as that of the dummy argument, unless the dummy argument is **assumed-rank**. The type parameter values of the **actual argument** shall agree with the corresponding ones of the dummy argument that are not **assumed** or **deferred**. The values of **assumed type parameters** of the **dummy argument** are assumed from the corresponding type parameters of its **effective argument**.
- 4 The **actual argument** shall have **deferred** the same type parameters as the dummy argument.

15.5.2.7 Allocatable dummy variables

- 1 The requirements in this subclause apply to **actual arguments** that correspond to **allocatable** dummy data objects.
- 2 The **actual argument** shall be **allocatable**. It is permissible for the **actual argument** to have an allocation status of unallocated.
- 3 The **corank** of the **actual argument** shall be the same as that of the **dummy argument**.
- 4 If the **actual argument** is a **coindexed object**, the dummy argument shall have the **INTENT (IN)** attribute.
- 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.

1 If such a **dummy argument** is used as an **actual argument** that is associated with a **dummy argument** with
 2 the **TARGET attribute**, whether any pointers associated with the original **actual argument** become associ-
 3 ated with the **dummy argument** with the **TARGET attribute** is processor dependent.

4 6 If the dummy argument has the **TARGET attribute**, does not have the **INTENT (OUT)** or **VALUE** attribute,
 5 and the corresponding **actual argument** has the **TARGET attribute** then

- 6 • any pointers associated with the **actual argument** become associated with the corresponding dummy
 7 argument on invocation of the procedure, and
- 8 • when execution of the procedure completes, any pointers that do not become undefined (**19.5.2.5**)
 9 and are associated with the dummy argument remain associated with the **actual argument**.

10 7 If a dummy argument has **INTENT (OUT)** or **INTENT (INOUT)**, the **actual argument** shall be **definable**. If a
 11 dummy argument has **INTENT (OUT)** and its associated **actual argument** is allocated, the **actual argument**
 12 is deallocated on procedure invocation (**9.7.3.2**).

13 15.5.2.8 Pointer dummy variables

14 1 The requirements in this subclause apply to **actual arguments** that correspond to dummy **data pointers**.

15 C1550 The **actual argument** corresponding to a dummy pointer with the **CONTIGUOUS attribute** shall be
 16 **simply contiguous** (**9.5.4**).

17 C1551 The **actual argument** corresponding to a dummy pointer shall not be a **coindexed object**.

NOTE 1

Constraint C1551 does not apply to any **intrinsic** procedure because an **intrinsic** procedure is defined in terms of its **actual arguments**.

18 2 If the dummy argument does not have **INTENT (IN)**, the **actual argument** shall be a pointer. Otherwise,
 19 the **actual argument** shall be a pointer or a valid **target** for the dummy pointer in a **pointer assignment**
 20 **statement**. If the **actual argument** is not a pointer, the dummy pointer becomes **pointer associated** with the
 21 **actual argument**.

22 3 If the dummy argument has **INTENT (OUT)**, the **pointer association** status of the **actual argument** becomes
 23 undefined on invocation of the procedure.

NOTE 2

For more explanatory information on pointers as dummy arguments, see **C.11.4**.

24 15.5.2.9 Coarray dummy variables

25 1 If the **dummy argument** is a **coarray**, the corresponding **actual argument** shall be a **coarray** and shall have
 26 the **VOLATILE attribute** if and only if the **dummy argument** has the **VOLATILE attribute**.

27 2 If the dummy argument is an array **coarray** that has the **CONTIGUOUS attribute** or is not of **assumed shape**,
 28 the corresponding **actual argument** shall be **simply contiguous** or an element of a **simply contiguous** array.

NOTE 1

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.

NOTE 2

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 3

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.

15.5.2.10 Actual arguments associated with dummy procedure entities

- 1 If the **interface** of a **dummy procedure** is **explicit**, its **characteristics** as a procedure (15.3.1) shall be the same as those of its **effective argument**, except that a **pure effective argument** may be associated with a dummy argument that is not **pure**, a **simple effective argument** may be associated with a dummy argument that is not **simple**, and an **elemental** intrinsic **actual** procedure¹³ 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 length¹⁴ 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

¹³Specific intrinsic procedure names are obsolescent.

¹⁴Assumed character length functions are obsolescent.

16.2¹⁵. 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)**; if the **actual argument** is not a **dummy argument** it becomes **pointer associated** with the **actual argument**, otherwise it becomes **pointer associated** with the **ultimate argument** of the **actual argument**.

6 When the **actual argument** is a procedure, the **host instance** of the **dummy argument** is the **host instance** of the **actual argument** (15.6.2.4).

7 If an **external procedure** or a **dummy procedure** is used as an **actual argument**, its **interface** shall be **explicit** or it shall be explicitly declared to have the **EXTERNAL attribute**.

15.5.2.11 Actual arguments and alternate return indicators (obsolescent)

1 If a dummy argument is an asterisk (15.6.2.3), the corresponding **actual argument** shall be an alternate return specifier (R1525).

2 Asterisk dummy arguments (alternate return indicators) and alternate return specifiers are obsolescent.

15.5.2.12 Sequence association

1 Sequence association only applies when the **dummy argument** is an **explicit-shape** or **assumed-size** array. The rest of this subclause only applies in that case.

2 An **actual argument** represents an element sequence if it is an array expression, an array element **designator**, a default character scalar, or a scalar of type character with the C character kind (18.2.2).

3 If the **dummy argument** is not of type character with default or C character kind:

- 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** of a **simply contiguous** array, the element sequence consists of that array element and each element that follows it in array element order;
- otherwise, if the **actual argument** is scalar, the element sequence consists of that scalar.

4 If the **dummy argument** is of type character with default or C character kind, and has nonzero character length, the storage unit sequence is as follows:

- if the actual argument is an array expression, the storage units of the array;
- if the actual argument is an array element or array element substring designator of a **simply contiguous** array, the storage units starting from the first storage unit of the designator and continuing to the end of the array;
- otherwise, if the actual argument is scalar, the storage units of the scalar object.

The element sequence is the sequence of consecutive groups of storage units in the storage unit sequence, grouped by the character length of the dummy array. The sequence terminates when the number of storage units left is less than the character length of the dummy array.

¹⁵Specific intrinsic procedure names are obsolescent.

NOTE

Some of the elements in the element sequence might consist of **storage units** from different elements of the original array.

- 1 5 If the **dummy argument** is of type character with default or C character kind, and has zero character length,
2 the element sequence consists of a sequence of elements each with zero character length, the number of
3 elements being the maximum number that is supported by the processor.
- 4 6 An **actual argument** that represents an element sequence and corresponds to a dummy argument that is an
5 array is sequence associated with the dummy argument. The **rank** and shape of the **actual argument** need
6 not agree with the **rank** and shape of the dummy argument, but the number of elements in the dummy
7 argument shall not exceed the number of elements in the element sequence of the **actual argument**. If the
8 dummy argument is **assumed-size**, the number of elements in the dummy argument is exactly the number
9 of elements in the element sequence.

10 15.5.2.13 Argument presence and restrictions on arguments not present

- 11 1 A **dummy argument** or an entity that is **host associated** with a **dummy argument** is not present if the **dummy**
12 **argument**
 - 13 • does not correspond to an **actual argument**,
 - 14 • corresponds to an **actual argument** that is not present, or
 - 15 • does not have the **ALLOCATABLE** or **POINTER** attribute, and corresponds to an **actual argument** that
 - 16 – has the **ALLOCATABLE attribute** and is not allocated, or
 - 17 – has the **POINTER attribute** and is **disassociated**;
- 18 otherwise, it is present.
- 19 2 A nonoptional **dummy argument** shall be present. If an optional nonpointer **dummy argument** corresponds
20 to a present pointer **actual argument**, the **pointer association** status of the **actual argument** shall not be
21 undefined.
- 22 3 An optional dummy argument that is not present is subject to the following restrictions.
 - 23 (1) If it is a data object, it shall not be referenced or be defined. If it is of a type that has default
24 initialization, the initialization has no effect.
 - 25 (2) It shall not be used as the **data-target** or **proc-target** of a **pointer assignment**.
 - 26 (3) If it is a procedure or **procedure pointer**, it shall not be invoked.
 - 27 (4) It shall not be supplied as an **actual argument** corresponding to a nonoptional dummy argument
28 other than as the argument of the intrinsic function **PRESENT** or as an argument of a function
29 reference that is a **constant expression**.
 - 30 (5) A **designator** with it as the **base object** and with one or more subobject selectors shall not be
31 supplied as an **actual argument**.
 - 32 (6) If it is an array, it shall not be supplied as an **actual argument** to an **elemental procedure** unless
33 an array of the same **rank** is supplied as an **actual argument** corresponding to a nonoptional
34 dummy argument of that **elemental procedure**.
 - 35 (7) If it is a pointer, it shall not be allocated, deallocated, nullified, pointer-assigned, or supplied as
36 an **actual argument** corresponding to an optional nonpointer dummy argument.
 - 37 (8) If it is **allocatable**, it shall not be allocated, deallocated, or supplied as an **actual argument** cor-
38 responding 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 a *selector* in an *ASSOCIATE*, *CHANGE TEAM*, *SELECT RANK*, or *SELECT TYPE* construct.
- (11) It shall not be supplied as the *data-ref* in a *procedure-designator*.
- (12) It 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.

15.5.2.14 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 of a subobject thereof, shall be taken through the dummy argument.
- (2) If the allocation status of the entity or a subobject thereof is affected through the dummy argument, then at any time during the invocation and execution of the procedure, either before or after the allocation or deallocation, it shall be referenced only through the dummy argument.
- (3) Action that affects the value of the entity or any subobject of it shall be taken only through the dummy argument unless
 - (a) the dummy argument has the *POINTER attribute*,
 - (b) the dummy argument is a scalar, *assumed-shape*, or *assumed-rank* object, and has the *TARGET attribute* but not the *INTENT (IN)* or *CONTIGUOUS* attributes, and the *actual argument* is a target other than a *coindexed object* or an *array section* with a *vector subscript*,
 - (c) the dummy argument is an *assumed-rank* object with the *TARGET attribute* and not the *INTENT (IN) attribute*, and the *actual argument* is a scalar target,
 - (d) the dummy argument is a *coarray* and the action is a coindexed definition of the corresponding *ultimate argument coarray* by a different *image*, or
 - (e) the dummy argument has a *coarray potential subobject component* and the action is a coindexed definition of the corresponding *coarray* by a different *image*.
- (4) If the value of the entity or any subobject of it is affected through the dummy argument, then at any time during the invocation and execution of the procedure, either before or after the definition, it shall be referenced only through that dummy argument unless
 - (a) the dummy argument has the *POINTER attribute*,
 - (b) the dummy argument is a scalar, *assumed-shape*, or *assumed-rank* object, and has the *TARGET attribute* but not the *INTENT (IN)* or *CONTIGUOUS* attributes, and the *actual argument* is a target other than a *coindexed object* or an *array section* with a *vector subscript*,
 - (c) the dummy argument is an *assumed-rank* object with the *TARGET attribute* and not the *INTENT (IN) attribute*, and the *actual argument* is a scalar target,
 - (d) the dummy argument is a *coarray* and the reference is a coindexed reference of its corresponding *ultimate argument coarray* by a different *image*, or
 - (e) the dummy argument has a *coarray potential subobject component* and the reference is a coindexed reference of the corresponding *coarray* by a different *image*.

NOTE 1

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,

```
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 2

If there is a partial or complete overlap between the [effective arguments](#) of two different dummy arguments of the same procedure and the dummy arguments have neither the [POINTER](#) nor [TARGET](#) attribute, the overlapped portions cannot be defined, redefined, or become undefined during the execution of the procedure. For example, in

```
CALL SUB (A (1:5), A (3:9))
```

the array section A (3:5) cannot be defined, redefined, or become undefined through the first dummy argument because it is part of the argument associated with the second dummy argument and cannot be defined, redefined, or become undefined through the second dummy argument because it is part of the argument associated with the first dummy argument. The array section A (1:2) remains [definable](#) through the first dummy argument and A (6:9) remains [definable](#) through the second dummy argument.

This restriction applies equally to pointer targets. In

```

REAL, DIMENSION (10), TARGET :: A
REAL, DIMENSION (:), POINTER :: B, C
B => A (1:5)
C => A (3:9)
CALL SUB (B, C) ! The dummy arguments of SUB are neither pointers nor targets.

```

the array section B (3:5) cannot be defined because it is part of the argument associated with the second dummy argument. The array section C (1:3) cannot be defined because it is part of the argument associated with the first

NOTE 2 (cont.)

dummy argument. The array section A (1:2), which is associated with B (1:2), remains [definable](#) through the first dummy argument and A (6:9), which is associated with C (4:7), remains [definable](#) through the second dummy argument.

NOTE 3

In

```

MODULE DATA
  REAL :: W, X, Y, Z
END MODULE DATA

PROGRAM MAIN
  USE DATA
  ...
  CALL INIT (X)
  ...
END PROGRAM MAIN
SUBROUTINE INIT (V)
  USE DATA
  ...
  READ (*, *) V
  ...
END SUBROUTINE INIT

```

variable X cannot be directly referenced at any time during the execution of INIT because it is being defined through the dummy argument V. X can be (indirectly) referenced through V. W, Y, and Z can be directly referenced. X can, of course, be directly referenced once execution of INIT is complete.

NOTE 4

The restrictions on entities associated with dummy arguments are intended to facilitate a variety of optimizations in the translation of the subprogram, including implementations of [argument association](#) in which the value of an [actual argument](#) that is neither a pointer nor a target is maintained in a register or in local storage.

NOTE 5

The exceptions to the aliasing restrictions for dummy arguments that are [coarrays](#) or have [coarray potential sub-object components](#) enable cross-image access while the procedure is executing. Because nonatomic accesses from different [images](#) typically need to be separated by an [image control statement](#), code optimization within segments is not unduly inhibited.

15.5.3 Function reference

- 1 A function is invoked during expression evaluation either by a [function-reference](#) or by a defined operation (10.1.6). When it is invoked, all [actual argument](#) expressions are evaluated, then the arguments are associated, and then the function is executed. When execution of the function is complete, the value of the function result is available for use in the expression that caused the function to be invoked. The [characteristics](#) of the function result (15.3.3) are determined by the [interface](#) of the function. If a reference to an [elemental function](#) (15.9) is an [elemental reference](#), all array arguments shall have the same shape.

15.5.4 Subroutine reference

- 1 A subroutine is invoked by execution of a [CALL statement](#), execution of a [defined assignment statement](#) (10.2.1.4), [defined input/output](#) (12.6.4.8.3), or [finalization](#) (7.5.6). When a subroutine is invoked, all [actual argument](#) expressions are evaluated, then the arguments are associated, and then the subroutine is executed. When the actions specified by the subroutine are completed, the execution of the [CALL statement](#), the execution of the [defined assignment statement](#), the processing of an [effective item](#), or [finalization](#) of an

object is also completed. If a **CALL statement** includes one or more alternate return specifiers¹⁶ among its arguments, a branch to one of the statements indicated might occur, depending on the action specified by the subroutine. If a reference to an **elemental** subroutine (15.9) is an **elemental reference**, at least one **actual argument** shall correspond to an **INTENT (OUT)** or **INTENT (INOUT)** dummy argument, all such **actual arguments** shall be arrays, and all **actual arguments** shall be **conformable**.

15.5.5 Resolving named procedure references

15.5.5.1 Establishment of procedure names

1 The rules for interpreting a procedure reference depend on whether the procedure name in the reference is established by the available declarations and specifications to be generic in the **scoping unit** containing the reference, is established to be only specific in the **scoping unit** containing the reference, or is not established.

2 A procedure name is established to be generic in a **scoping unit**

- (1) if that **scoping unit** contains an **interface block** with that name;
- (2) if that **scoping unit** contains a **GENERIC statement** with a *generic-spec* that is that name;
- (3) if that **scoping unit** contains an **INTRINSIC attribute** specification for that name and it is the generic name of an intrinsic procedure;
- (4) if that **scoping unit** contains a **USE statement** that makes that procedure name accessible and the corresponding name in the module is established to be generic; or
- (5) if that **scoping unit** contains no declarations of that name, that **scoping unit** has a **host scoping unit**, and that name is established to be generic in the **host scoping unit**.

3 A procedure name is established to be only specific in a **scoping unit** if it is established to be specific and not established to be generic. It is established to be specific

- (1) if that **scoping unit** contains a **module subprogram**, **internal subprogram**, or **statement function statement**¹⁷ 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 procedure¹⁸;
- (4) if that **scoping unit** contains an explicit **EXTERNAL attribute** specification for that name;
- (5) if that **scoping unit** contains a **USE statement** that makes that procedure name accessible and the corresponding name in the module is established to be specific; or
- (6) if that **scoping unit** contains no declarations of that name, that **scoping unit** has a **host scoping unit**, and that name is established to be specific in the **host scoping unit**.

4 A procedure name is not established in a **scoping unit** if it is neither established to be generic nor established to be specific.

15.5.5.2 Resolving procedure references to names established to be generic

1 If the reference is consistent with a non**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 procedure in the

¹⁶Alternate return specifiers are obsolescent.

¹⁷Statement functions are obsolescent.

¹⁸Specific intrinsic procedure names are obsolescent.

- 1 [interface block](#) that provides that [interface](#). The rules in [15.4.3.4.5](#) ensure that there can be at most one
2 such specific procedure.
- 3 2 Otherwise, if the reference is consistent with an [elemental reference](#) to one of the [specific interfaces](#) of a
4 [generic interface](#) that has that name and either is defined in the [scoping unit](#) in which the reference appears
5 or is made accessible by a [USE statement](#) in the [scoping unit](#), the reference is to the specific [elemental](#)
6 [procedure](#) in the [interface block](#) that provides that [interface](#). The rules in [15.4.3.4.5](#) ensure that there can
7 be at most one such specific [elemental procedure](#).
- 8 3 Otherwise, if the [scoping unit](#) contains either an [INTRINSIC attribute](#) specification for that name or a [USE](#)
9 [statement](#) that makes that name accessible from a module in which the corresponding name is specified to
10 have the [INTRINSIC attribute](#), and if the reference is consistent with the [interface](#) of that [intrinsic](#) proced-
11 ure, the reference is to that [intrinsic](#) procedure.
- 12 4 Otherwise, if the [scoping unit](#) has a [host scoping unit](#), the name is established to be generic in that [host](#)
13 [scoping unit](#), and there is agreement between the [scoping unit](#) and the [host scoping unit](#) as to whether the
14 name is a function name or a subroutine name, the name is resolved by applying the rules in this subclause
15 to the [host scoping unit](#) as if the reference appeared there.
- 16 5 Otherwise, if the name is that of an intrinsic procedure and the reference is consistent with that intrinsic
17 procedure, the reference is to that intrinsic procedure.

NOTE 1

Because of the renaming facility of the [USE statement](#), the name in the reference can be different from the usual name of the intrinsic procedure.

NOTE 2

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(:)
    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.

18 **15.5.5.3 Resolving procedure references to names established to be only specific**

- 19 1 If the name has the [EXTERNAL attribute](#),
- 20 • if it is a [procedure pointer](#), the reference is to its target;
- 21 • if it is a [dummy procedure](#) that is not a [procedure pointer](#), the reference is to the [effective argument](#)
- 22 corresponding to that name;

- otherwise, the reference is to the [external procedure](#) with that name.

- If the name is that of an accessible [external procedure](#), [internal procedure](#), [module procedure](#), [intrinsic procedure](#), or statement function¹⁹, the reference is to that procedure.

NOTE

Because of the renaming facility of the [USE statement](#), the name in the reference can be different from the original name of the procedure.

15.5.5.4 Resolving procedure references to names not established

- 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](#).
- 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.
- Otherwise, the reference is to an [external procedure](#) with that name.

15.5.6 Resolving type-bound procedure references

- If the *binding-name* in a [procedure-designator](#) (R1522) is that of a specific [type-bound procedure](#), the procedure referenced is the one bound to that name in the [dynamic type](#) of the [data-ref](#).
- 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.
- The reference is to the procedure bound to the same name as the selected specific [binding](#) in the [dynamic type](#) of the [data-ref](#).

15.6 Procedure definition**15.6.1 Intrinsic procedure definition**

- Intrinsic procedures are defined as an inherent part of the processor. A standard-conforming processor shall include the intrinsic procedures described in Clause 16, but may include others. However, a standard-conforming program shall not make use of intrinsic procedures other than those described in Clause 16.

¹⁹Statement functions are obsolescent.

15.6.2 Procedures defined by subprograms

15.6.2.1 General

1 A procedure is defined by the initial **SUBROUTINE** or **FUNCTION** statement of a subprogram, and each **ENTRY statement**²⁰ defines an additional procedure (15.6.2.6).

2 A subprogram is specified to have the **NON_RECURSIVE** attribute, or to be **elemental** (15.9), **pure** (15.7), or a separate module subprogram (15.6.2.5) by a *prefix* in its initial **SUBROUTINE** or **FUNCTION** statement.

R1529 *prefix* **is** *prefix-spec* [*prefix-spec*] ...

R1530 *prefix-spec* **is** *declaration-type-spec*

or ELEMENTAL

or IMPURE

or MODULE

or NON_RECURSIVE

or PURE

or RECURSIVE

or SIMPLE

C1552 (R1529) A *prefix* shall contain at most one of each *prefix-spec*.

C1553 (R1529) A *prefix* that specifies IMPURE shall specify neither PURE nor SIMPLE.

C1554 (R1529) A *prefix* shall not specify both NON_RECURSIVE and RECURSIVE.

C1555 An **elemental procedure** shall not have the **BIND** attribute.

C1556 (R1529) **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.

C1557 (R1529) 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.

C1558 (R1529) 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**.

C1559 (R1529) 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**.

C1560 (R1529) 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. If a subprogram defines a function whose name is declared with an asterisk *type-param-value*²¹, no procedure defined by the subprogram shall directly or indirectly invoke itself or any other procedure defined by the subprogram. The **RECURSIVE**

²⁰The **ENTRY statement** is obsolescent.

²¹That is, an assumed character length function; such functions are obsolescent.

1 *prefix-spec* is advisory only.

2 4 If the *prefix-spec* PURE or the *prefix-spec* SIMPLE appears, or the *prefix-spec* ELEMENTAL appears and IM-
3 PURE does not appear, the subprogram is a *pure* subprogram and shall meet the additional constraints of
4 15.7. If the *prefix-spec* SIMPLE appears, the subprogram is a *simple* subprogram and shall meet the addi-
5 tional constraints of 15.8.

6 5 If the *prefix-spec* ELEMENTAL appears, the subprogram is an *elemental subprogram* and shall meet the
7 additional constraints of 15.9.1.

8 R1531 *proc-language-binding-spec* is *language-binding-spec*

9 6 A *proc-language-binding-spec* specifies that the procedure defined or declared by the statement is interop-
10 erable (18.3.7).

11 C1561 A *proc-language-binding-spec* with a *NAME= specifier* shall not be specified in the *function-stmt*
12 or *subroutine-stmt* of an *internal procedure*, or of an *interface body* for an *abstract interface* or a
13 *dummy procedure*.

14 C1562 If *proc-language-binding-spec* is specified for a function, the function result shall be an *interop-*
15 *erable scalar variable*.

16 C1563 If *proc-language-binding-spec* is specified for a procedure, each of its dummy arguments shall be
17 an *interoperable* procedure (18.3.7) or a variable that is *interoperable* (18.3.5, 18.3.6), *assumed-*
18 *shape*, *assumed-rank*, *assumed-type*, of type CHARACTER with assumed length, or that has the
19 *ALLOCATABLE* or *POINTER* attribute.

20 C1564 If *proc-language-binding-spec* is specified for a procedure, each dummy argument of type CHAR-
21 ACTER with the *ALLOCATABLE* or *POINTER* attribute shall have *deferred* character length.

22 C1565 A variable that is a dummy argument of a procedure that has a *proc-language-binding-spec* shall be
23 *assumed-type* or of *interoperable* type and *kind type parameters*.

24 C1566 If *proc-language-binding-spec* is specified for a procedure, it shall not have a *default-initialized*
25 *dummy argument* with the *ALLOCATABLE* or *POINTER* attribute.

26 C1567 If *proc-language-binding-spec* is specified for a procedure, it shall not have a dummy argument that
27 is a *coarray*.

28 C1568 If *proc-language-binding-spec* is specified for a procedure, it shall not have an array dummy argu-
29 ment with the *VALUE* attribute.

30 15.6.2.2 Function subprogram

31 1 A function subprogram is a subprogram that has a FUNCTION statement as its first statement.

32 R1532 *function-subprogram* is *function-stmt*
33 [*specification-part*]
34 [*execution-part*]
35 [*internal-subprogram-part*]
36 *end-function-stmt*

37 R1533 *function-stmt* is [*prefix*] FUNCTION *function-name* ■
38 ■ ([*dummy-arg-name-list*]) [*suffix*]

- 1 C1569 (R1533) If **RESULT** appears, *result-name* shall not be the same as *function-name* and shall not be
2 the same as the *entry-name* in any **ENTRY statement**²² in the subprogram.
- 3 C1570 (R1533) If **RESULT** appears, the *function-name* shall not appear in any specification statement in
4 the **scoping unit** of the function subprogram.
- 5 R1534 *dummy-arg-name* **is** *name*
- 6 C1571 (R1534) A *dummy-arg-name* shall be the name of a dummy argument.
- 7 R1535 *suffix* **is** *proc-language-binding-spec* [**RESULT** (*result-name*)]
8 **or** **RESULT** (*result-name*) [*proc-language-binding-spec*]
- 9 R1536 *end-function-stmt* **is** **END** [**FUNCTION** [*function-name*]]
- 10 C1572 (R1532) An internal function subprogram shall not contain an *internal-subprogram-part*.
- 11 C1573 (R1536) If a *function-name* appears in the *end-function-stmt*, it shall be identical to the *function-*
12 *name* specified in the *function-stmt*.
- 13 2 The name of the function is *function-name*.
- 14 3 The type and type parameters (if any) of the result of the function defined by a function subprogram may be
15 specified by a type specification in the **FUNCTION** statement or by the name of the **function result** appearing
16 in a **type declaration statement** in the specification part of the function subprogram. They shall not be
17 specified both ways. If they are not specified either way, they are determined by the implicit typing rules in
18 effect within the function subprogram. If the **function result** is an array, **allocatable**, or a pointer, this shall
19 be specified by specifications of the name of the **function result** within the function body. The specifications
20 of the **function result** attributes, the specification of dummy argument attributes, and the information in
21 the **FUNCTION** statement collectively define the **characteristics** of the function (15.3.1).
- 22 4 If **RESULT** appears, the name of the **function result** of the function is *result-name* and all occurrences of
23 the function name in *execution-part* statements in its scope refer to the function itself. If **RESULT** does
24 not appear, the name of the **function result** is *function-name* and all occurrences of the function name in
25 *execution-part* statements in its scope are references to the **function result**. On completion of execution of
26 the function, the value returned is that of its **function result**. If the **function result** is a **data pointer**, the shape
27 of the value returned by the function is determined by the shape of the **function result** when the execution
28 of the function is completed. If the **function result** is not a pointer, its value shall be defined by the function.
29 If the **function result** is a pointer, on return the **pointer association** status of the **function result** shall not be
30 undefined.

NOTE 1

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 2

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
```

²²The **ENTRY statement** is obsolescent.

NOTE 2 (cont.)

```

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);
```

15.6.2.3 Subroutine subprogram

1 A subroutine subprogram is a subprogram that has a SUBROUTINE statement as its first statement.

R1537 *subroutine-subprogram* **is** *subroutine-stmt*
 [*specification-part*]
 [*execution-part*]
 [*internal-subprogram-part*]
 end-subroutine-stmt

R1538 *subroutine-stmt* **is** [*prefix*] SUBROUTINE *subroutine-name* ■
 ■ [([*dummy-arg-list*]) [*proc-language-binding-spec*]]

C1574 (R1538) The *prefix* of a *subroutine-stmt* shall not contain a *declaration-type-spec*.

R1539 *dummy-arg* **is** *dummy-arg-name*
 or *

R1540 *end-subroutine-stmt* **is** END [SUBROUTINE [*subroutine-name*]]

C1575 (R1537) An internal subroutine subprogram shall not contain an *internal-subprogram-part*.

C1576 If a *subroutine-name* appears in an *end-subroutine-stmt*, it shall be identical to the *subroutine-name* specified in the subprogram's *subroutine-stmt*.

2 The name of the subroutine is *subroutine-name*.

15.6.2.4 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 [unsaved local procedure pointers](#). Saved local entities are shared by all instances of the subprogram.

2 When a statement function²³ is invoked, an instance of that statement function is created.

3 When execution of an instance completes it ceases to exist.

²³Statement functions are obsolescent.

1 4 The caller of an instance of a procedure is the instance of the main program, subprogram, or statement
 2 function that invoked it. The call sequence of an instance of a procedure is its caller, followed by the call
 3 sequence of its caller. The call sequence of the main program is empty. The **host instance** of an instance
 4 of a statement function or an **internal procedure** that is invoked by its name is the first element of the call
 5 sequence that is an instance of the host of the statement function or internal subprogram. The **host instance**
 6 of an **internal procedure** that is invoked via a **dummy procedure** or **procedure pointer** is the **host instance** of
 7 the **associating entity** from when the **argument association** or **pointer association** was established (19.5.5).
 8 The **host instance** of a **module procedure** is the module or submodule in which it is defined. A main program
 9 or external subprogram has no **host instance**.

10 15.6.2.5 Separate module procedures

11 1 A separate module procedure is a **module procedure** defined by a *separate-module-subprogram*, by a *func-*
 12 *tion-subprogram* whose initial statement contains the keyword MODULE, or by a *subroutine-subprogram*
 13 whose initial statement contains the keyword MODULE.

14 R1541 *separate-module-subprogram* **is** *mp-subprogram-stmt*
 15 [*specification-part*]
 16 [*execution-part*]
 17 [*internal-subprogram-part*]
 18 *end-mp-subprogram-stmt*

19 R1542 *mp-subprogram-stmt* **is** MODULE PROCEDURE *procedure-name*

20 R1543 *end-mp-subprogram-stmt* **is** END [PROCEDURE [*procedure-name*]]

21 C1577 (R1541) The *procedure-name* shall have been declared to be a separate **module procedure** in the
 22 containing **program unit** or an ancestor of that **program unit**.

23 C1578 (R1543) If a *procedure-name* appears in the *end-mp-subprogram-stmt*, it shall be identical to the
 24 *procedure-name* in the *mp-subprogram-stmt*.

25 2 A separate **module procedure** shall not be defined more than once.

26 3 The **interface** of a procedure defined by a *separate-module-subprogram* is explicitly declared by the *mp-*
 27 *subprogram-stmt* to be the same as its module procedure **interface body**. It has the **NON_RECURSIVE at-**
 28 **tribute** if and only if it was declared to have that attribute by the **interface body**. If it is a function its result
 29 name is determined by the **FUNCTION statement** in the **interface body**.

NOTE

A separate **module procedure** can be accessed by **use association** only if its **interface body** is declared in the spe-
 cification part of a module and is public.

30 15.6.2.6 ENTRY statement (obsolescent)

31 1 An ENTRY statement permits a procedure reference to begin with a particular executable statement within
 32 the function or subroutine subprogram in which the ENTRY statement appears.

33 2 The ENTRY statement is obsolescent.

34 R1544 *entry-stmt* **is** ENTRY *entry-name* [([*dummy-arg-list*]) [*suffix*]]

35 C1579 (R1544) If **RESULT** appears, the *entry-name* shall not appear in any specification or **type declara-**

- 1 tion statement in the *scoping unit* of the function subprogram.
- 2 C1580 (R1544) An *entry-stmt* shall appear only in an *external-subprogram* or a *module-subprogram* that
 3 does not define a separate *module procedure*. An *entry-stmt* shall not appear within an *executable-*
 4 *construct*.
- 5 C1581 (R1544) **RESULT** shall appear only if the *entry-stmt* is in a function subprogram.
- 6 C1582 (R1544) A *dummy-arg* shall not be an alternate return indicator if the **ENTRY** statement is in a
 7 function subprogram.
- 8 C1583 (R1544) If **RESULT** appears, *result-name* shall not be the same as the *function-name* in the **FUNC-**
 9 **TION statement** and shall not be the same as the *entry-name* in any **ENTRY** statement in the sub-
 10 program.
- 11 3 Optionally, a subprogram may have one or more **ENTRY** statements.
- 12 4 If the **ENTRY** statement is in a function subprogram, an additional function is defined by that subprogram.
 13 The name of the function is *entry-name* and the name of its *result* is *result-name* or is *entry-name* if no *result-*
 14 *name* is provided. The *dummy arguments* of the function are those specified in the **ENTRY** statement. If the
 15 *characteristics* of the result of the function named in the **ENTRY** statement are the same as the *character-*
 16 *istics* of the result of the function named in the **FUNCTION statement**, their *result* names identify the same
 17 entity, although their names need not be the same. Otherwise, they are *storage associated* and shall all
 18 be nonpointer, nonallocatable scalar variables that are default integer, default real, double precision real,
 19 default complex, or default logical.
- 20 5 If the **ENTRY** statement is in a subroutine subprogram, an additional subroutine is defined by that sub-
 21 program. The name of the subroutine is *entry-name*. The dummy arguments of the subroutine are those
 22 specified in the **ENTRY** statement.
- 23 6 The order, number, types, kind type parameters, and names of the dummy arguments in an **ENTRY** state-
 24 ment may differ from the order, number, types, kind type parameters, and names of the dummy arguments
 25 in the **FUNCTION** or **SUBROUTINE** statement in the containing subprogram.
- 26 7 Because an **ENTRY** statement defines an additional function or an additional subroutine, it is referenced in
 27 the same manner as any other function or subroutine (15.5).
- 28 8 In a subprogram, a dummy argument specified in an **ENTRY** statement shall not appear in an executable
 29 statement preceding that **ENTRY** statement, unless it also appears in a **FUNCTION**, **SUBROUTINE**, or **ENTRY**
 30 statement that precedes the executable statement. A function result specified by a *result-name* in an **ENTRY**
 31 statement shall not appear in any executable statement that precedes the first **RESULT** clause with that
 32 name.
- 33 9 In a subprogram, a *dummy argument* specified in an **ENTRY** statement shall not appear in the expression
 34 of a *statement function* that precedes the first *dummy-arg* with that name in the subprogram. A function
 35 result specified by a *result-name* in an **ENTRY** statement shall not appear in the expression of a *statement*
 36 *function* that precedes the first **RESULT** clause with that name.
- 37 10 If a *dummy argument* appears in an executable statement, the execution of the executable statement is
 38 permitted during the execution of a reference to the function or subroutine only if the *dummy argument*
 39 appears in the *dummy argument* list of the referenced procedure.
- 40 11 If a *dummy argument* is used in a *specification expression* to specify an array bound or character length

of an object, the appearance of the object in a statement that is executed during a procedure reference is permitted only if the [dummy argument](#) appears in the [dummy argument](#) list of the referenced procedure and it is present ([15.5.2.13](#)).

The [NON_RECURSIVE](#) and [RECURSIVE](#) keywords are not used in an ENTRY statement. Instead, the presence or absence of [NON_RECURSIVE](#) in the initial [SUBROUTINE](#) or [FUNCTION](#) statement controls whether the procedure defined by an ENTRY statement is permitted to reference itself or another procedure defined by the subprogram.

The keywords [PURE](#) and [IMPURE](#) are not used in an ENTRY statement. Instead, the procedure defined by an ENTRY statement is [pure](#) if and only if the subprogram is a [pure](#) subprogram.

The keyword [ELEMENTAL](#) is not used in an ENTRY statement. Instead, the procedure defined by an ENTRY statement is [elemental](#) if and only if [ELEMENTAL](#) is specified in the [SUBROUTINE](#) or [FUNCTION](#) statement.

15.6.2.7 RETURN statement

R1545 *return-stmt* is RETURN [*scalar-int-expr*]

C1584 (R1545) The *return-stmt* shall be in the [inclusive scope](#) of a function or subroutine subprogram.

C1585 (R1545) 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 ([11.2](#)) to the [branch target statement](#) identified by the n^{th} alternate return specifier in the [actual argument](#) list of the referenced procedure. Such an expression is also known as an alternate return indicator; these are obsolescent. 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.

15.6.2.8 CONTAINS statement

R1546 *contains-stmt* is CONTAINS

1 The CONTAINS statement separates the body of a main program, module, submodule, or subprogram from any internal or module subprograms it might contain, or it introduces the type-bound procedure part of a derived-type definition ([7.5.5](#)). The CONTAINS statement is not executable.

15.6.3 Definition and invocation of procedures by means other than Fortran

1 A procedure may be defined by means other than Fortran. The [interface](#) of a procedure defined by means other than Fortran may be specified by an [interface body](#) or [procedure declaration statement](#). A reference to such a procedure is made as though it were defined by an external subprogram.

2 A procedure defined by means other than Fortran that is invoked by a Fortran procedure and does not cause [termination of execution](#) shall return to its caller.

NOTE 1

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 document.

- 1 3 If the [interface](#) of a procedure has a *proc-language-binding-spec*, the procedure is [interoperable](#) (18.10).
 2 4 Interoperation with C functions is described in 18.10.

NOTE 2

For explanatory information on definition of procedures by means other than Fortran, see [C.11.2](#).

15.6.4 Statement function (obsolescent)

- 1 A statement function is a function defined by a single statement. Statement functions are obsolescent.
- R1547 *stmt-function-stmt* **is** *function-name* ([*dummy-arg-name-list*]) = *scalar-expr*
- C1586 (R1547) 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.
- C1587 (R1547) [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.
- C1588 (R1547) 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.
- C1589 (R1547) The *function-name* and each *dummy-arg-name* shall be specified, explicitly or implicitly, to be scalar.
- C1590 (R1547) A given *dummy-arg-name* shall not appear more than once in a given *dummy-arg-name-list*.
- C1591 A statement function shall not be of a parameterized derived type.
- 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.

15.7 Pure procedures

1 A **pure procedure** is

- a **simple procedure**,
- a pure intrinsic procedure (16.1),
- a **module procedure** in an intrinsic module, if it is specified to be pure,
- defined by a pure subprogram,
- a **dummy procedure** that has been specified to be **PURE**,
- a **procedure pointer** that has been specified to be **PURE**,
- a **type-bound procedure** that is bound to a pure procedure, or
- a statement function²⁴ that references only pure functions and does not contain the *designator* of a variable with the **VOLATILE** attribute.

2 A pure subprogram is a subprogram that has the *prefix-spec* **PURE** or the *prefix-spec* **SIMPLE**, or that has the *prefix-spec* **ELEMENTAL** and does not have the *prefix-spec* **IMPURE**. The following additional constraints apply to pure subprograms.

C1592 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.

C1593 The function result of a pure function shall not be such that **finalization** of a reference to the function would reference an impure procedure.

C1594 The function result of a pure function shall not be both **polymorphic** and allocatable, or have a **polymorphic** allocatable **ultimate component**.

C1595 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.

C1596 An **INTENT (OUT)** **dummy argument** of a pure procedure shall not be such that **finalization** of the actual argument would reference an impure procedure.

C1597 An **INTENT (OUT)** **dummy argument** of a pure procedure shall not be **polymorphic** or have a **polymorphic** allocatable **ultimate component**.

C1598 A **local variable** of a pure subprogram, or of a **BLOCK construct** within a pure subprogram, shall not have the **SAVE** or **VOLATILE** attribute.

NOTE 1

Variable initialization in a *type-declaration-stmt* or a *data-stmt* implies the **SAVE** attribute; therefore, such initialization is also disallowed.

C1599 A named local entity or construct entity of a pure subprogram shall not be of a type that has default initialization of a data pointer component to a target at any level of component selection.

C15100 The *specification-part* of a pure subprogram shall specify that all its **dummy procedures** are pure.

C15101 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.

²⁴Statement functions are obsolescent.

1 C15102 All internal subprograms in a pure subprogram shall be pure.

2 C15103 A *designator* of a variable with the VOLATILE attribute shall not appear in a pure subprogram.

3 C15104 In a pure subprogram any *designator* with a *base object* that is in common or accessed by *use* or
4 *host* association, is a *pointer dummy argument* of a pure function, is a *dummy argument* with the
5 *INTENT (IN) attribute*, is a *coindexed object*, or an object that is *storage associated*²⁵ with any such
6 variable, shall not be used

7 (1) in a variable definition context (19.6.7),

8 (2) in a pointer association context (19.6.8),

9 (3) as the *data-target* in a *pointer-assignment-stmt*,

10 (4) as the *expr* corresponding to a component in a *structure-constructor* if the component has
11 the *POINTER attribute* or has a pointer component at any level of component selection,

12 (5) as the *expr* of an *intrinsic assignment statement* in which the variable is of a derived type if
13 the derived type has a pointer component at any level of component selection,

14 (6) as the *source-expr* in a *SOURCE= specifier* if the designator is of a derived type that has a
15 pointer component at any level of component selection,

16 (7) as an *actual argument* corresponding to a dummy argument with the *POINTER attribute*, or

17 (8) as the *actual argument* to the function *C_LOC* from the intrinsic module *ISO_C_BINDING*.

NOTE 2

Item 5 requires that processors be able to determine if entities with the *PRIVATE attribute* or with private components have a pointer component.

18 C15105 Any procedure referenced in a pure subprogram, including one referenced via a *defined operation*,
19 *defined assignment*, *defined input/output*, or *finalization*, shall be pure.

20 C15106 A statement that might result in the deallocation of a *polymorphic* entity is not permitted in a
21 pure procedure.

NOTE 3

This includes *intrinsic assignment* to a variable that has a *potential subobject component* that is *polymorphic* and *allocatable*.

22 C15107 A pure subprogram shall not contain a *print-stmt*, *open-stmt*, *close-stmt*, *backspace-stmt*, *endfile-*
23 *stmt*, *rewind-stmt*, *flush-stmt*, *wait-stmt*, or *inquire-stmt*.

24 C15108 A pure subprogram shall not contain a *read-stmt* or *write-stmt* whose *io-unit* is a *file-unit-number*
25 or *.

26 C15109 A pure subprogram shall not contain an *image control statement* (11.7.1).

27 C15110 A reference to the function *C_FUNLOC* from the intrinsic module *ISO_C_BINDING* shall not appear
28 in a pure subprogram if its argument is impure.

NOTE 4

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.

²⁵Common blocks and storage association are obsolescent.

NOTE 4 (cont.)

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 cannot 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 can a pure subprogram contain any operation that could conceivably perform any [external file](#) input/output or execute an image control statement (including a [STOP statement](#)). 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 5

Pure subroutines are included to allow subroutine calls from [pure procedures](#) in a safe way, and to allow [forall-assignment-stmts](#) to be [defined assignments](#). The constraints for pure subroutines are based on the same principles as for pure functions, except that side effects to [INTENT \(OUT\)](#), [INTENT \(INOUT\)](#), and pointer dummy arguments are permitted.

15.8 Simple procedures

1 A [simple procedure](#) is

- an intrinsic procedure ([16.1](#)), if it is specified to be simple,
- a [module procedure](#), if it is specified to be simple,
- a procedure defined by a simple subprogram,
- a [dummy procedure](#) that has been specified to be simple,
- a [procedure pointer](#) that has been specified to be simple,
- a [type-bound procedure](#) that is bound to a simple procedure,
- a deferred [type-bound procedure](#) whose interface specifies it to be simple,
- a statement function²⁶ defined in a simple subprogram.

2 A simple procedure is also a [pure procedure](#) and is subject to the constraints for pure procedures ([15.7](#)). A simple procedure can also be an [elemental procedure](#).

3 A simple subprogram is a subprogram that has the [prefix-spec](#) SIMPLE. The following additional constraints apply to simple subprograms.

C15111 The [specification-part](#) of a simple subprogram shall specify that all of its [dummy procedures](#) are simple.

C15112 If a procedure that is not an intrinsic procedure, a [module procedure](#) of an intrinsic module, or a statement function is used in a context that requires it to be simple, then its interface shall be explicit in the scope of that use. The interface shall specify that the procedure is simple.

C15113 All internal subprograms in a simple subprogram shall be simple.

C15114 Any procedure referenced in a simple subprogram shall be simple.

²⁶Statement functions are obsolescent.

C15115 A simple subprogram shall not contain a designator of a variable that is accessed by `use` or `host` association, unless the designator is part of a specification inquiry (10.1.11) that is a constant expression.

C15116 A simple subprogram shall not contain a reference to a variable in a common block²⁷.

C15117 A simple subprogram shall not contain an `ENTRY statement`²⁸.

15.9 Elemental procedures

15.9.1 Elemental procedure declaration and interface

1 An `elemental procedure` is

- an elemental `intrinsic procedure` (16.1),
- a `module procedure` in an `intrinsic module`, if it is specified to be elemental,
- a procedure that is defined by an `elemental subprogram`, or
- a `type-bound procedure` that is bound to an elemental procedure.

An `elemental procedure` has only `scalar dummy arguments`, but may have array `actual arguments`.

2 A `dummy procedure` or `procedure pointer` shall not be specified to be `ELEMENTAL`.

3 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`.

C15118 All `dummy arguments` of an `elemental procedure` shall be scalar noncoarray dummy data objects and shall not have the `POINTER` or `ALLOCATABLE` attribute.

C15119 The `result` of an `elemental function` shall be scalar, and shall not have the `POINTER` or `ALLOCATABLE` attribute.

C15120 The `specification-part` of an `elemental subprogram` shall specify the intents of all of its `dummy arguments` that do not have the `VALUE` attribute.

C15121 In the `specification-expr` that specifies a `type parameter` value of the result of an elemental function, an `object designator` with a `dummy argument` of the function as the `base object` shall appear only as the subject of a specification inquiry (10.1.11), and that specification inquiry shall not depend on a property that is deferred.

4 In a reference to an elemental procedure, if any argument is an array, each `actual argument` that corresponds to an `INTENT (OUT)` or `INTENT (INOUT) dummy argument` shall be an array. All `actual arguments` shall be `conformable`. An array `actual argument` is considered to be associated with the scalar `dummy arguments` of the procedure throughout the entire execution of the elemental reference; thus, the restrictions on actions specified in 15.5.2.14 apply to the entirety of the `actual array argument`.

15.9.2 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,

²⁷Common blocks are obsolescent.

²⁸The `ENTRY statement` is obsolescent.

- 1 of the result are the same as would have been obtained if the scalar function had been applied separately,
 2 in array element order, to corresponding elements of each array [actual argument](#).

NOTE

An example of an [elemental reference](#) to the intrinsic function [MAX](#): if X and Y are arrays with bounds (1:M, 1:N), then

[MAX](#) (X, 0.0, Y)

is an array expression of shape [M, N] whose elements in order have the values of

[([MAX](#) (X(I, J), 0.0, Y(I, J)), I = 1, M), J = 1, N)]

3 **15.9.3 Elemental subroutine actual arguments**

- 4 1 In a reference to an elemental subroutine, if the [actual arguments](#) corresponding to [INTENT \(OUT\)](#) and [IN-](#)
 5 [TENT \(INOUT\) dummy arguments](#) are arrays, the values of the elements, if any, of the results are the same as
 6 would be obtained if the subroutine had been applied separately, in array element order, to corresponding
 7 elements of each array [actual argument](#).

16 Intrinsic procedures and modules

16.1 Classes of intrinsic procedures

- 1 Intrinsic procedures are divided into eight classes: [inquiry functions](#), [elemental functions](#), [transformational functions](#), [elemental subroutines](#), [simple subroutines](#), [atomic subroutines](#), [collective 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 [15.9](#). 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 [simple](#).
- 3 An [atomic subroutine](#) is an intrinsic subroutine that performs an atomic action. The semantics of atomic actions are described in [16.5](#).
- 4 A [collective subroutine](#) is an intrinsic subroutine that performs a cooperative calculation on a [team of images](#) without requiring synchronization. The semantics of [collective subroutines](#) are described in [16.6](#).
- 5 The subroutine [MOVE_ALLOC](#) with noncoarray argument FROM, the subroutine [SPLIT](#), the subroutine [TOKENIZE](#), and the [elemental](#) subroutine [MVBITS](#), are [simple](#). No other standard intrinsic subroutine is [pure](#) or [simple](#).
- 6 [Generic names](#) of standard intrinsic procedures are listed in [16.7](#). In most cases, generic functions accept arguments of more than one type and the type of the result is the same as the type of the arguments. [Specific names](#)¹ of standard intrinsic functions with corresponding generic names are listed in [16.8](#).
- 7 If an intrinsic procedure is used as an [actual argument](#) to a procedure, its [specific name](#) shall be used and it shall be referenced in the called procedure only with scalar arguments. If an intrinsic procedure does not have a [specific name](#), it shall not be used as an [actual argument](#) ([15.5.2.10](#)).
- 8 [Elemental](#) intrinsic procedures behave as described in [15.9](#).

16.2 Arguments to intrinsic procedures

16.2.1 General rules

- 1 All intrinsic procedures can be invoked with either positional arguments or [argument keywords](#) ([15.5](#)). The descriptions in [16.7](#) through [16.9](#) give the [argument keyword](#) names and positional sequence for standard intrinsic procedures.
- 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

¹Specific intrinsic procedure names are obsolescent.

in square brackets in description headings and in lists of procedures. The valid forms of reference for procedures with optional arguments are described in 15.5.2.

NOTE 1

The text CMPLX (X [, Y, KIND]) indicates that Y and KIND are both optional arguments. Valid reference forms include CMPLX(x), CMPLX(x,y), CMPLX(x, KIND=kind), CMPLX(x,y, kind), and CMPLX(KIND=kind, X=x, Y=y).

NOTE 2

Some intrinsic procedures impose additional requirements on their optional arguments. For example, [SELECTED-REAL_KIND](#) requires that at least one of its optional arguments be present, and [RANDOM_SEED](#) requires that at most one of its optional arguments be present.

3 The dummy arguments of the specific intrinsic procedures² in 16.8 have [INTENT \(IN\)](#). The dummy arguments of the intrinsic procedures in 16.9 have [INTENT \(IN\)](#) if the intent is not stated explicitly.

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](#) (10.2.1.3).

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 15.5.2.14.

7 An argument to an intrinsic procedure other than [ASSOCIATED](#), [NULL](#), or [PRESENT](#) shall be a data object.

16.2.2 The shape of array arguments

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.

16.2.3 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.

16.2.4 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.

²Specific intrinsic procedure names are obsolescent.

- 1 2 The process of reducing a dimension usually combines the selected elements with a simple operation such
 2 as addition or an intrinsic function such as **MAX**, but more sophisticated reductions are also provided, e.g.
 3 by **COUNT** and **MAXLOC**.

4 **16.3 Bit model**

5 **16.3.1 General**

- 6 1 The bit manipulation procedures are described in terms of a model for the representation and behavior of
 7 bits on a processor.
- 8 2 For the purposes of these procedures, a bit is defined to be a binary digit w located at position k of a non-
 9 negative integer scalar object based on a model nonnegative integer defined by

$$j = \sum_{k=0}^{z-1} w_k \times 2^k$$

10 and for which w_k has the value 0 or 1. This defines a sequence of bits $w_{z-1} \dots w_0$, with w_{z-1} the leftmost
 11 bit and w_0 the rightmost bit. The positions of bits in the sequence are numbered from right to left, with
 12 the position of the rightmost bit being zero. The length of a sequence of bits is z . An example of a model
 13 number compatible with the examples used in 16.4 would have $z = 32$, thereby defining a 32-bit integer.

14 3 The interpretation of a negative integer as a sequence of bits is processor dependent.

15 4 The **inquiry function** **BIT_SIZE** provides the value of the parameter z of the model.

16 5 Effectively, this model defines an integer object to consist of z bits in sequence numbered from right to left
 17 from 0 to $z - 1$. This model is valid only in the context of the use of such an object as the argument or
 18 result of an intrinsic procedure that interprets that object as a sequence of bits. In all other contexts, the
 19 model defined for an integer in 16.4 applies. In particular, whereas the models are identical for $r = 2$ and
 20 $w_{z-1} = 0$, they do not correspond for $r \neq 2$ or $w_{z-1} = 1$ and the interpretation of bits in such objects is
 21 processor dependent.

22 **16.3.2 Bit sequence comparisons**

23 1 When bit sequences of unequal length are compared, the shorter sequence is considered to be extended to
 24 the length of the longer sequence by padding with zero bits on the left.

25 2 Bit sequences are compared from left to right, one bit at a time, until unequal bits are found or all bits have
 26 been compared and found to be equal. If unequal bits are found, the sequence with zero in the unequal pos-
 27 ition is considered to be less than the sequence with one in the unequal position. Otherwise the sequences
 28 are considered to be equal.

29 **16.3.3 Bit sequences as arguments to INT and REAL**

- 30 1 When a **boz-literal-constant** is the argument A of the intrinsic function **INT** or **REAL**,
- 31 • if the length of the sequence of bits specified by A is less than the size in bits of a scalar variable of
 32 the same type and kind type parameter as the result, the **boz-literal-constant** is treated as if it were
 33 extended to a length equal to the size in bits of the result by padding on the left with zero bits, and
 - 34 • if the length of the sequence of bits specified by A is greater than the size in bits of a scalar variable of
 35 the same type and kind type parameter as the result, the **boz-literal-constant** is treated as if it were
 36 truncated from the left to a length equal to the size in bits of the result.

- 1 C1601 If a *boz-literal-constant* is truncated as an argument to the intrinsic function *REAL*, the discarded
2 bits shall all be zero.

NOTE

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*.

3 16.4 Numeric models

- 4 1 The numeric manipulation and *inquiry functions* are described in terms of a model for the representation
5 and behavior of numbers on a processor. The model has parameters that are determined so as to make the
6 model best fit the machine on which the program is executed.

- 7 2 The model set for integer i is defined by

$$i = s \times \sum_{k=0}^{q-1} w_k \times r^k$$

- 8 where r is an integer exceeding one, q is a positive integer, each w_k is a nonnegative integer less than r , and
9 s is +1 or −1. The integer parameters r and q determine the set of model integers.

- 10 3 The model set for real x is defined by

$$x = \begin{cases} 0 \text{ or} \\ s \times b^e \times \sum_{k=1}^p f_k \times b^{-k} \end{cases}$$

- 11 where b and p are integers exceeding one; each f_k is a nonnegative integer less than b , with f_1 nonzero; s
12 is +1 or −1; and e is an integer that lies between some integer maximum $e_{\max}$ and some integer minimum
13 $e_{\min}$ inclusively. For $x = 0$, its exponent e and digits f_k are defined to be zero. The integer parameters b , p ,
14 $e_{\min}$, and $e_{\max}$ determine the set of model floating-point numbers.

- 15 4 The parameters of the integer and real models are available for each representation method of the integer
16 and real types. The parameters characterize the set of available numbers in the definition of the model.
17 Intrinsic functions provide the values of some parameters and other values related to the models.

- 18 5 There is also an extended model set for each kind of real x ; this extended model is the same as the ordinary
19 model except that there are no limits on the range of the exponent e .

NOTE

Some of the function descriptions use the models

$$i = s \times \sum_{k=0}^{30} w_k \times 2^k$$

and

$$x = 0 \text{ or } s \times 2^e \times \left(\frac{1}{2} + \sum_{k=2}^{24} f_k \times 2^{-k} \right), \quad -126 \leq e \leq 127$$

16.5 Atomic subroutines

- 1 An **atomic subroutine** is an intrinsic subroutine that performs an action on its ATOM argument, and if it has an OLD argument, determination of the value to be assigned to that argument, atomically. Definition or evaluation of any argument other than ATOM is not performed atomically.
- 2 For any two executions in unordered segments of atomic subroutines whose ATOM argument is the same object, the effect is as if one of the executions is performed completely before the other execution begins. Which execution is performed first is processor dependent. The sequence of **atomic** actions within ordered **segments** is specified in 5.3.5. If successive atomic subroutine invocations on image P redefine a variable atomically in **segments** P_i and P_j , atomic references to that variable from image Q in a **segment** Q_k that is unordered relative to P_i and P_j may observe the changes in the value of that variable in any order.
- 3 Atomic operations shall make asynchronous progress. If a variable X on image P is defined by an **atomic subroutine** on image Q , image R repeatedly references $X[P]$ by an atomic subroutine in an unordered **segment**, and no other image defines $X[P]$ in an unordered **segment**, image R shall eventually receive the value assigned by image Q , even if none of the images P , Q , or R execute an **image control statement** until after the definition of $X[P]$ by image Q and the reception of that value by image R .
- 4 If the STAT argument is present in an invocation of an atomic subroutine and no error condition occurs, it is assigned the value zero.
- 5 If the STAT argument is present in an invocation of an atomic subroutine and an error condition occurs, any other argument that is not **INTENT (IN)** becomes undefined. If the ATOM argument is on a failed image, an error condition occurs and the value **STAT_FAILED_IMAGE** from the intrinsic module **ISO_FORTRAN_ENV** is assigned to the STAT argument. If any other error condition occurs, the STAT argument is assigned a processor-dependent positive value that is different from the value of **STAT_FAILED_IMAGE**.
- 6 If the STAT argument is not present in an invocation of an atomic subroutine and an error condition occurs, **error termination** is initiated.

NOTE

The properties of atomic subroutines are intended to support custom synchronization mechanisms. The program will need to handle all possible orderings of sequences of atomic subroutine executions that can arise as a consequence of the above rules; note that the orderings can appear to be different on different images even in the same program execution.

16.6 Collective subroutines

- 1 Successful execution of a **collective subroutine** performs a calculation on all the **images** of the **current team** and assigns a computed value on one or all of them. If it is invoked by one **image**, it shall be invoked by the same statement on all **active images** of its **current team** in **segments** that are not ordered with respect to each other; corresponding references participate in the same collective computation.
 - 2 Before execution of the first **CHANGE TEAM statement** on an **image**, in between executions of **CHANGE TEAM** and/or **END TEAM** statements, and after the last execution of an **END TEAM** statement, the sequence of invocations of **collective subroutines** shall be the same on all **active images** of a **team**. A **collective subroutine** shall not be referenced when an **image control statement** is not permitted to be executed (for example, in a procedure invoked from a **CRITICAL construct**).
- C1602 A reference to a **collective subroutine** shall not appear in a context where an **image control statement** is not permitted to appear.

- 1 3 If the A argument in a reference to a **collective subroutine** is a **coarray**, the corresponding **ultimate arguments** on all **active images** of the **current team** shall be corresponding **coarrays** as described in 5.4.7.
- 2
- 3 4 If the STAT argument is present in a reference to a **collective subroutine** on one image:
- 4
- 5 • it shall be present on all the corresponding references;
 - 6 • if no error condition occurs on that image, it is assigned the value zero;
 - 7 • if an error condition occurs on that **image**, the A argument becomes undefined;
 - 8 • if an error condition occurs other than that an **image** in the **current team** has **stopped** or **failed**, the
- 9 STAT argument is assigned a processor-dependent positive value that is different from the value of **STAT_STOPPED_IMAGE** or **STAT_FAILED_IMAGE** from the intrinsic module **ISO_FORTRAN_ENV**.
- 10 5 A reference to a **collective subroutine** on an image may be successful even if an error condition occurs
- 11 during the corresponding reference on another image. If error conditions occur on more than one image,
- 12 the error conditions may be different.
- 13 6 If the **current team** contains an **image** that is known to have **stopped**, an error condition occurs, and if the
- 14 STAT argument is present it is assigned the value **STAT_STOPPED_IMAGE** from the intrinsic module **ISO_**
- 15 **FORTRAN_ENV**. Otherwise, if the **current team** contained an image that is known to have **failed**, an error
- 16 condition occurs, and if the STAT argument is present it is assigned the value **STAT_FAILED_IMAGE** from
- 17 the intrinsic module **ISO_FORTRAN_ENV**.
- 18 7 If the STAT argument is not present in a reference to a **collective subroutine** and an error condition occurs,
- 19 error termination is initiated.
- 20 8 If the ERRMSG argument is present in a reference to a **collective subroutine** and an error condition occurs, it
- 21 is assigned an explanatory message, as if by **intrinsic assignment**. If no error condition occurs, the definition
- 22 status and value of ERRMSG are unchanged.

NOTE 1

The argument A becomes undefined if an error condition occurs during execution of a **collective subroutine** because it is intended to allow the processor to use A for intermediate values during calculation.

NOTE 2

Although the calculations performed by a collective subroutine have some internal synchronizations, a reference to a collective subroutine is not an **image control statement**.

23 16.7 Standard generic intrinsic procedures

- 24 1 For all of the standard intrinsic procedures, the arguments shown are the names that shall be used for
- 25 **argument keywords** if the keyword form is used for **actual arguments**.

NOTE 1

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 2

Many of the **argument keywords** have names that are indicative of their usage. For example:

NOTE 2 (cont.)

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

- 1 2 In the Class column of Table 16.1,
- 3 A indicates that the procedure is an **atomic subroutine**,
- 4 C indicates that the procedure is a **collective subroutine**,
- 5 E indicates that the procedure is an **elemental function**,
- 6 ES indicates that the procedure is a **simple elemental subroutine**,
- 7 I indicates that the procedure is an **inquiry function**,
- 8 PS indicates that the procedure is a **simple subroutine** when the FROM argument is not a **coarray**,
- 9 S indicates that the procedure is an impure subroutine,
- 10 SS indicates that the procedure is a **simple subroutine**, and
- T indicates that the procedure in a **transformational function**.

Table 16.1 — 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.
ACOSD (X)	E	Arc cosine function in degrees.
ACOSH (X)	E	Inverse hyperbolic cosine function.
ACOSPI (X)	E	Circular arc 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 ALL (MASK, DIM)	T	Array reduced by .AND. operator.
ALLOCATED (ARRAY) or ALLOCATED (SCALAR)	I	Allocation status of allocatable variable.
ANINT (A [, KIND])	E	Nearest whole number.
ANY (MASK) or ANY (MASK, DIM)	T	Array reduced by .OR. operator.
ASIN (X)	E	Arcsine (inverse sine) function.
ASIND (X)	E	Arc sine function in degrees.
ASINH (X)	E	Inverse hyperbolic sine function.
ASINPI (X)	E	Circular arc sine function.
ASSOCIATED (POINTER [, TARGET])	I	Pointer association status inquiry.
ATAN (X) or ATAN (Y, X)	E	Arctangent (inverse tangent) function.
ATAN2 (Y, X)	E	Arctangent (inverse tangent) function.
ATAN2D (Y, X)	E	Arc tangent function in degrees.
ATAN2PI (Y, X)	E	Circular arc tangent function.
ATAND (X) or ATAND (Y, X)	E	Arc tangent function in degrees.
ATANH (X)	E	Inverse hyperbolic tangent function.
ATANPI (X) or ATANPI (Y, X)	E	Circular arc tangent function.
ATOMIC_ADD (ATOM, VALUE [, STAT])	A	Atomic addition.
ATOMIC_AND (ATOM, VALUE [, STAT])	A	Atomic bitwise AND.
ATOMIC_CAS (ATOM, OLD, COMPARE, NEW[, STAT])	A	Atomic compare and swap.
ATOMIC_DEFINE (ATOM, VALUE [, STAT])	A	Define a variable atomically.
ATOMIC_FETCH_ADD (ATOM, VALUE, OLD [, STAT])	A	Atomic fetch and add.
ATOMIC_FETCH_AND (ATOM, VALUE, OLD [, STAT])	A	Atomic fetch and bitwise AND.
ATOMIC_FETCH_OR (ATOM, VALUE, OLD [, STAT])	A	Atomic fetch and bitwise OR.
ATOMIC_FETCH_XOR (ATOM, VALUE, OLD [, STAT])	A	Atomic fetch and bitwise exclusive OR.
ATOMIC_OR (ATOM, VALUE [, STAT])	A	Atomic bitwise OR.
ATOMIC_REF (VALUE, ATOM [, STAT])	A	Reference a variable atomically.

Table 16.1: Standard generic intrinsic procedure summary (cont.)

Procedure (arguments)	Class	Description
ATOMIC_XOR (ATOM, VALUE [, STAT])	A	Atomic bitwise exclusive OR.
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 16.3 .
BLE (I, J)	E	Bitwise less than or equal to.
BLT (I, J)	E	Bitwise less than.
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 CMPLX (X [, Y, KIND])	E	Conversion to complex type.
CO_BROADCAST (A, SOURCE_IMAGE [, STAT, ERRMSG])	C	Broadcast value to images.
CO_MAX (A [, RESULT_IMAGE, STAT, ERRMSG])	C	Compute maximum value across images.
CO_MIN (A [, RESULT_IMAGE, STAT, ERRMSG])	C	Compute minimum value across images.
CO_REDUCE (A, OPERATION [, RESULT_IMAGE, STAT, ERRMSG])	C	Generalized reduction across images.
CO_SUM (A [, RESULT_IMAGE, STAT, ERRMSG])	C	Compute sum across images.
COMMAND_ARGUMENT_COUNT ()	T	Number of command arguments.
CONJG (Z)	E	Conjugate of a complex number.
COS (X)	E	Cosine function.
COSD (X)	E	Degree cosine function.
COSH (X)	E	Hyperbolic cosine function.
COSHAPE (COARRAY [, KIND])	I	Sizes of codimensions of a coarray.
COSPI (X)	E	Circular cosine function.
COUNT (MASK [, DIM, KIND])	T	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.
EVENT_QUERY (EVENT, COUNT [, STAT])	S	Query event count.
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.
FAILED_IMAGES ([TEAM, KIND])	T	Indices of failed images.
FINDLOC (ARRAY, VALUE [, MASK, KIND, BACK]) or FINDLOC (ARRAY, VALUE, DIM [, 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, ERRMSG])	S	Get program invocation command.
GET_COMMAND_ARGUMENT (NUMBER [, VALUE, LENGTH, STATUS, ERRMSG])	S	Get program invocation argument.
GET_ENVIRONMENT_VARIABLE (NAME [, VALUE, LENGTH,	S	Get environment variable.

Table 16.1: Standard generic intrinsic procedure summary (cont.)

Procedure (arguments)	Class	Description
STATUS, TRIM_NAME, ERRMSG])		
GET_TEAM ([LEVEL])	T	Team.
HUGE (X)	I	Largest model value or last enumeration value.
HYPOT (X, Y)	E	Euclidean distance function.
IACHAR (C [, KIND])	E	ASCII code value for character.
IALL (ARRAY, DIM [, MASK]) or IALL (ARRAY [, MASK])	T	Array reduced by IAND function.
IAND (I, J)	E	Bitwise AND.
IANY (ARRAY, DIM [, MASK]) or IANY (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, TEAM_NUMBER) or IMAGE_INDEX (COARRAY, SUB, TEAM) or IMAGE_INDEX (COARRAY, SUB)	T	Image index from cosubscripts .
IMAGE_STATUS (IMAGE [, TEAM])	E	Image execution state.
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 IPARITY (ARRAY [, MASK])	T	Array reduced by IEOOR 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 (8.5.7).
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).
LCOBUND (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 MAXLOC (ARRAY [, MASK, KIND, BACK])	T	Location(s) of maximum value.
MAXVAL (ARRAY, DIM [, MASK]) or MAXVAL (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.
MINLOC (ARRAY, DIM [, MASK, KIND, BACK]) or MINLOC (ARRAY [, MASK, KIND, BACK])	T	Location(s) of minimum value.
MINVAL (ARRAY, DIM [, MASK]) or MINVAL (ARRAY [, MASK])	T	Minimum value(s) of array.
MOD (A, P)	E	Remainder function.
MODULO (A, P)	E	Modulo function.
MOVE_ALLOC (FROM, TO [, STAT, ERRMSG])	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.
NEXT (A [, STAT])	E	Next enumeration value.
NINT (A [, KIND])	E	Nearest integer.

Table 16.1: Standard generic intrinsic procedure summary (cont.)

Procedure (arguments)	Class	Description
NORM2 (X) or NORM2 (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 () or NUM_IMAGES (TEAM) or NUM_IMAGES (TEAM_NUMBER)	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 PARITY (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.
PREVIOUS (A [, STAT])	E	Previous enumeration value.
PRODUCT (ARRAY, DIM [, MASK]) or PRODUCT (ARRAY [, MASK])	T	Array reduced by multiplication.
RADIX (X)	I	Base of a numeric model.
RANDOM_INIT (REPEATABLE, IMAGE_DISTINCT)	S	Initialize pseudorandom number generator.
RANDOM_NUMBER (HARVEST)	S	Generate pseudorandom number(s).
RANDOM_SEED ([SIZE, PUT, GET])	S	Pseudorandom number generator control.
RANGE (X)	I	Decimal exponent range of a numeric model (16.4).
RANK (A)	I	Rank of a data object.
REAL (A [, KIND])	E	Conversion to real type.
REDUCE (ARRAY, OPERATION [, MASK, IDENTITY, ORDERED]) or REDUCE (ARRAY, OPERATION, DIM [, 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.
SELECTED_CHAR_KIND (NAME)	T	Character kind selection.
SELECTED_INT_KIND (R)	T	Integer kind selection.
SELECTED_LOGICAL_KIND (BITS)	T	Logical 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.
SIND (X)	E	Degree sine function.
SINH (X)	E	Hyperbolic sine function.
SINPI (X)	E	Circular sine function.
SIZE (ARRAY [, DIM, KIND])	I	Size of an array or one extent.
SPACING (X)	E	Spacing of model numbers.
SPLIT (STRING, SET, POS [, BACK])	SS	Parse a string into tokens, one at a time.
SPREAD (SOURCE, DIM, NCOPIES)	T	Value replicated in a new dimension.
SQRT (X)	E	Square root.
STOPPED_IMAGES ([TEAM, KIND])	T	Indices of stopped images.
STORAGE_SIZE (A [, KIND])	I	Storage size in bits.
SUM (ARRAY, DIM [, MASK]) or SUM (ARRAY [, MASK])	T	Array reduced by addition.
SYSTEM_CLOCK ([COUNT, COUNT_RATE, COUNT_MAX])	S	Query system clock.
TAN (X)	E	Tangent function.
TAND (X)	E	Degree tangent function.
TANH (X)	E	Hyperbolic tangent function.
TANPI (X)	E	Circular tangent function.
TEAM_NUMBER ([TEAM])	T	Team number.
THIS_IMAGE ([TEAM])	T	Index of the invoking image .
THIS_IMAGE (COARRAY [, TEAM]) or	T	Cosubscript(s) for this image .

Table 16.1: Standard generic intrinsic procedure summary (cont.)

Procedure (arguments)	Class	Description
THIS_IMAGE (COARRAY, DIM [, TEAM])		
TINY (X)	I	Smallest positive model number.
TOKENIZE (STRING, SET, TOKENS [, SEPARATOR]) or TOKENIZE (STRING, SET, FIRST, LAST)	SS	Parse a string into tokens.
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.

1 3 The effect of calling [EXECUTE_COMMAND_LINE](#) on any [image](#) other than [image](#) 1 in the [initial team](#) is pro-
2 cessor dependent.

3 4 The use of all other standard intrinsic procedures in unordered segments is subject only to their argument
4 use following the rules in [11.7.2](#).

5 16.8 Specific names for standard intrinsic functions (obsolescent)

6 1 Specific names for standard intrinsic functions are obsolescent.

7 2 Except for [AMAX0](#), [AMIN0](#), [MAX1](#), and [MIN1](#), the result type of the specific function is the same that the
8 result type of the corresponding generic function reference would be if it were invoked with the same
9 arguments as the specific function.

10 3 A function listed in [Table 16.3](#) is not permitted to be used as an [actual argument](#) ([15.5.1](#), [C1534](#)), as a
11 target in a procedure [pointer assignment statement](#) ([10.2.2.2](#), [C1033](#)), as an initial target in a [procedure](#)
12 [declaration statement](#) ([15.4.3.6](#), [C1519](#)), or to specify an interface ([15.4.3.6](#), [C1515](#)).

Table 16.2 — 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

Unrestricted specific intrinsic functions (cont.)

Specific name	Generic name	Argument type and kind
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 16.3 — 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

16.9 Specifications of the standard intrinsic procedures

16.9.1 General

1 Detailed specifications of the standard generic intrinsic procedures are provided in 16.9 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 the 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 When an allocatable [deferred-length](#) character scalar corresponding to an [INTENT \(INOUT\)](#) or [INTENT \(OUT\)](#) argument is assigned a value, the value is assigned as if by [intrinsic assignment](#).

4 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.

5 The result values of some functions are described using pseudo-subscripts (s_1 to s_n) of the argument array(s). These should be interpreted as if the lower bounds of the arrays were all equal to one.

16.9.2 ABS (A)

1 **Description.** Absolute value.

2 **Class.** [Elemental](#) function.

3 **Argument.** A shall be of type integer, real, or complex.

4 **Result Characteristics.** The same as A except that if A is complex, the result is real.

5 **Result Value.** If A is of type integer or real, the value of the result is $|A|$; if A is complex with value (x, y) , the result is equal to a processor-dependent approximation to $\sqrt{x^2 + y^2}$ computed without undue overflow or underflow.

6 **Example.** ABS ((3.0, 4.0)) has the value 5.0 (approximately).

16.9.3 ACHAR (I [, KIND])

1 **Description.** Character from ASCII code value.

2 **Class.** [Elemental](#) function.

3 **Arguments.**

I shall be of type integer.

KIND (optional) shall be a scalar integer [constant expression](#).

1 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.

3 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.

7 6 **Example.** ACHAR (88) has the value 'X'.

8 16.9.4 ACOS (X)

9 1 **Description.** Arccosine (inverse cosine) function.

10 2 **Class.** [Elemental](#) function.

11 3 **Argument.** X shall be of type real with a value that satisfies the inequality $|X| \leq 1$, or of type complex.

12 4 **Result Characteristics.** Same as X.

13 5 **Result Value.** The result has a value equal to a processor-dependent approximation to $\arccos(X)$. If it is real it is expressed in radians and lies in the range $0 \leq \text{ACOS}(X) \leq \pi$. If it is complex the real part is expressed in radians and lies in the range $0 \leq \text{REAL}(\text{ACOS}(X)) \leq \pi$.

16 6 **Example.** ACOS (0.54030231) has the value 1.0 (approximately).

17 16.9.5 ACOSD (X)

18 1 **Description.** Arc cosine function in degrees.

19 2 **Class.** [Elemental](#) function.

20 3 **Argument.** X shall be of type real with a value that satisfies the inequality $|X| \leq 1$.

21 4 **Result Characteristics.** Same as X.

22 5 **Result Value.** The result has a value equal to a processor-dependent approximation to the arc cosine of X. It is expressed in degrees and lies in the range $0 \leq \text{ACOSD}(X) \leq 180$.

24 6 **Example.** ACOSD (−1.0) has the value 180.0 (approximately).

25 16.9.6 ACOSH (X)

26 1 **Description.** Inverse hyperbolic cosine function.

27 2 **Class.** [Elemental](#) function.

28 3 **Argument.** X shall be of type real or complex.

29 4 **Result Characteristics.** Same as X.

30 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 real part is nonnegative, and the imaginary part is expressed in radians and lies in the range $-\pi \leq \text{AIMAG}(\text{ACOSH}(X)) \leq \pi$

33 6 **Example.** ACOSH (1.5430806) has the value 1.0 (approximately).

16.9.7 ACOSPI (X)

Description. Circular arc cosine function.

Class. [Elemental](#) function.

Argument. X shall be of type real with a value that satisfies the inequality $|X| \leq 1$.

Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to the arc cosine of X. It is expressed in half-revolutions and lies in the range $0 \leq \text{ACOSPI}(X) \leq 1$.

Example. ACOSPI (−1.0) has the value 1.0 (approximately).

16.9.8 ADJUSTL (STRING)

Description. Left-justified string value.

Class. [Elemental](#) function.

Argument. STRING shall be of type character.

Result Characteristics. Character of the same length and kind type parameter as STRING.

Result Value. The value of the result is the same as STRING except that any leading blanks have been deleted and the same number of trailing blanks have been inserted.

Example. ADJUSTL (' WORD') has the value 'WORD '.

16.9.9 ADJUSTR (STRING)

Description. Right-justified string value.

Class. [Elemental](#) function.

Argument. STRING shall be of type character.

Result Characteristics. Character of the same length and kind type parameter as STRING.

Result Value. The value of the result is the same as STRING except that any trailing blanks have been deleted and the same number of leading blanks have been inserted.

Example. ADJUSTR ('WORD ') has the value ' WORD'.

16.9.10 AIMAG (Z)

Description. Imaginary part of a complex number.

Class. [Elemental](#) function.

Argument. Z shall be of type complex.

Result Characteristics. Real with the same kind type parameter as Z.

Result Value. If Z has the value (x, y) , the result has the value y .

1 6 **Example.** AIMAG ((2.0, 3.0)) has the value 3.0.

2 16.9.11 AINT (A [, KIND])

3 1 **Description.** Truncation toward 0 to a whole number.

4 2 **Class.** Elemental function.

5 3 **Arguments.**

6 A shall be of type real.

7 KIND (optional) shall be a scalar integer constant expression.

8 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.

10 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.

13 6 **Examples.** AINT (2.783) has the value 2.0. AINT (−2.783) has the value −2.0.

14 16.9.12 ALL (MASK) or ALL (MASK, DIM)

15 1 **Description.** Array reduced by .AND. operator.

16 2 **Class.** Transformational function.

17 3 **Arguments.**

18 MASK shall be a logical array.

19 DIM shall be an integer scalar with value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of MASK.

20 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 $[d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n]$ where $[d_1, d_2, \dots, d_n]$ is the shape of MASK.

23 5 **Result Value.**

24 *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.

26 *Case (ii):* If MASK has rank one, ALL (MASK, DIM) is equal to ALL (MASK). Otherwise, the value of element $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$ of ALL (MASK, DIM) is equal to ALL (MASK $(s_1, s_2, \dots, s_{\text{DIM}-1}, \vdots, s_{\text{DIM}+1}, \dots, s_n)$).

29 6 **Examples.**

30 *Case (i):* The value of ALL ([.TRUE., .FALSE., .TRUE.]) is false.

31 *Case (ii):* If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ and C is the array $\begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}$ then ALL (B /= C, DIM = 1) is [true, false, false] and ALL (B /= C, DIM = 2) is [false, false].

16.9.13 ALLOCATED (ARRAY) or ALLOCATED (SCALAR)

Description. Allocation status of allocatable variable.

Class. Inquiry function.

Arguments.

ARRAY shall be an allocatable array.

SCALAR shall be an allocatable scalar.

Result Characteristics. Default logical scalar.

Result Value. The result has the value true if the argument (ARRAY or SCALAR) is allocated and has the value false if the argument is unallocated.

16.9.14 ANINT (A [, KIND])

Description. Nearest whole number.

Class. Elemental function.

Arguments.

A shall be of type real.

KIND (optional) shall be a scalar integer constant expression.

Result Characteristics. The result is of type real. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of A.

Result Value. The result is the integer nearest A, or if there are two integers equally near A, the result is whichever such integer has the greater magnitude.

Examples. ANINT (2.783) has the value 3.0. ANINT (−2.783) has the value −3.0.

16.9.15 ANY (MASK) or ANY (MASK, DIM)

Description. Array reduced by .OR. operator.

Class. Transformational function.

Arguments.

MASK shall be a logical array.

DIM shall be an integer scalar with a value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of MASK.

Result Characteristics. The result is of type logical with the same kind type parameter as MASK. It is scalar if DIM does not appear or $n = 1$; otherwise, the result has rank $n - 1$ and shape $[d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n]$ where $[d_1, d_2, \dots, d_n]$ is the shape of MASK.

Result Value.

Case (i): The result of ANY (MASK) has the value true if any element of MASK is true and has the value false if no elements are true or if MASK has size zero.

Case (ii): If MASK has rank one, ANY (MASK, DIM) is equal to ANY (MASK). Otherwise, the value of element $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$ of ANY (MASK, DIM) is equal to ANY (MASK $(s_1, s_2, \dots,$

$s_{\text{DIM}-1}, \dots, s_{\text{DIM}+1}, \dots, s_n$)).

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].

16.9.16 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 $|X| \leq 1$, 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 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 $-\pi/2 \leq \text{REAL}(\text{ASIN}(X)) \leq \pi/2$.

6 **Example.** ASIN (0.84147098) has the value 1.0 (approximately).

16.9.17 ASIND (X)

1 **Description.** Arc sine function in degrees.

2 **Class.** Elemental function.

3 **Argument.** X shall be of type real with a value that satisfies the inequality $|X| \leq 1$.

4 **Result Characteristics.** Same as X.

5 **Result Value.** The result has a value equal to a processor-dependent approximation to the arc sine of X. It is expressed in degrees and lies in the range $-90 \leq \text{ASIND}(X) \leq 90$.

6 **Example.** ASIND (1.0) has the value 90.0 (approximately).

16.9.18 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 $-\pi/2 \leq \text{AIMAG}(\text{ASINH}(X)) \leq \pi/2$.

6 **Example.** ASINH (1.1752012) has the value 1.0 (approximately).

16.9.19 ASINPI (X)

Description. Circular arc sine function.

Class. [Elemental](#) function.

Argument. X shall be of type real with a value that satisfies the inequality $|X| \leq 1$.

Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to the arc sine of X. It is expressed in half-revolutions and lies in the range $-\frac{1}{2} \leq \text{ASINPI}(X) \leq \frac{1}{2}$.

Example. ASINPI (1.0) has the value 0.5 (approximately).

16.9.20 ASSOCIATED (POINTER [, TARGET])

Description. [Pointer association](#) status inquiry.

Class. [Inquiry function](#).

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 a pointer or an entity that could be a [target](#). If TARGET is a pointer then its [pointer association](#) status shall not be undefined.

If POINTER is a [procedure pointer](#), TARGET shall be a procedure (or [procedure pointer](#)) that would be allowable as the [target](#) of a [pointer assignment](#) (10.2.2) for a [procedure pointer](#) with the same [characteristics](#) as POINTER.

Otherwise, TARGET shall be a noncoindexed variable that is not an array section with a vector subscript, or a reference to a function that returns a data pointer. If POINTER is not [unlimited polymorphic](#), TARGET shall be [type compatible](#) with it, and the corresponding [kind type parameters](#) shall be equal. If POINTER is not [assumed-rank](#), TARGET shall have the same rank as POINTER.

Result Characteristics. Default logical scalar.

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 other than a [dummy procedure](#) or [procedure pointer](#), the result is true if and only if POINTER is [associated](#) with TARGET and, if TARGET is an [internal procedure](#), they have the same [host instance](#).

Case (iii): If TARGET is present and is a [dummy procedure](#) that is not a [procedure pointer](#), the result is true if and only if POINTER is [associated](#) with the procedure that is the [ultimate argument](#) of TARGET and, if the procedure is an [internal procedure](#), they have the same [host instance](#).

Case (iv): If TARGET is present and is a [procedure pointer](#), the result is true if and only if POINTER and TARGET are [associated](#) with the same procedure and, if the procedure is an [internal procedure](#), they have the same [host instance](#).

Case (v): 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 (vi): 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 (vii): 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 (viii): 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

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 (iv), (vii), or (viii).

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.

16.9.21 ATAN (X) or ATAN (Y, X)

Description. Arctangent (inverse tangent) function.

Class. Elemental function.

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.

Result Characteristics. Same as X.

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 $-\pi/2 \leq \text{ATAN}(X) \leq \pi/2$.

Example. ATAN (1.5574077) has the value 1.0 (approximately).

16.9.22 ATAN2 (Y, X)

Description. Arctangent (inverse tangent) function.

Class. Elemental function.

Arguments.

Y shall be of type real.

1 X shall be of the same type and kind type parameter as Y. If Y has the value zero, X shall not have
2 the value zero.

3 4 **Result Characteristics.** Same as X.

4 5 **Result Value.** The result has a value equal to a processor-dependent approximation to the principal value
5 of the argument of the complex number (X, Y), expressed in radians. It lies in the range $-\pi \leq \text{ATAN2}(Y, X)$
6 $\leq \pi$ and is equal to a processor-dependent approximation to a value of $\arctan(Y/X)$ if $X \neq 0$. If $Y > 0$, the
7 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
8 if Y is positive real zero or the processor does not distinguish between positive and negative real zero, and
9 approximately $-\pi$ if Y is negative real zero. If $Y < 0$, the result is negative. If $X = 0$, the absolute value of
10 the result is approximately $\pi/2$.

11 6 **Examples.** $\text{ATAN2}(1.5574077, 1.0)$ has the value 1.0 (approximately). If Y has the value $\begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix}$ and
12 X has the value $\begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}$, the value of $\text{ATAN2}(Y, X)$ is approximately $\begin{bmatrix} 3\pi/4 & \pi/4 \\ -3\pi/4 & -\pi/4 \end{bmatrix}$.

13 16.9.23 ATAN2D (Y, X)

14 1 **Description.** Arc tangent function in degrees.

15 2 **Class.** [Elemental](#) function.

16 3 **Arguments.**

17 Y shall be of type real.

18 X shall be of the same type and kind type parameter as Y. If Y has the value zero, X shall not have
19 the value zero.

20 4 **Result Characteristics.** Same as X.

21 5 **Result Value.** The result is expressed in degrees and lies in the range $-180 \leq \text{ATAN2D}(Y, X) \leq 180$. It has
22 a value equal to a processor-dependent approximation to $\text{ATAN2}(Y, X) \times 180/\pi$.

23 6 **Examples.** $\text{ATAN2D}(1.0, 1.0)$ has the value 45.0 (approximately). If Y has the value $\begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix}$ and X has
24 the value $\begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}$, the value of $\text{ATAN2D}(Y, X)$ is approximately $\begin{bmatrix} 135.0 & 45.0 \\ -135.0 & -45.0 \end{bmatrix}$.

25 16.9.24 ATAN2PI (Y, X)

26 1 **Description.** Circular arc tangent function.

27 2 **Class.** [Elemental](#) function.

28 3 **Arguments.**

29 Y shall be of type real.

30 X shall be of the same type and kind type parameter as Y. If Y has the value zero, X shall not have
31 the value zero.

32 4 **Result Characteristics.** Same as X.

1 5 **Result Value.** The result is expressed in half-revolutions and lies in the range $-1 \leq \text{ATAN2PI}(Y, X) \leq 1$. It
 2 has a value equal to a processor-dependent approximation to $\text{ATAN2}(Y, X) \div \pi$.

3 6 **Examples.** $\text{ATAN2PI}(1.0, 1.0)$ has the value 0.25 (approximately). If Y has the value $\begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix}$ and X
 4 has the value $\begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}$, the value of $\text{ATAN2PI}(Y, X)$ is approximately $\begin{bmatrix} 0.75 & 0.25 \\ -0.75 & -0.25 \end{bmatrix}$.

5 16.9.25 ATAND (X) or ATAND (Y, X)

6 1 **Description.** Arc tangent function in degrees.

7 2 **Class.** [Elemental](#) function.

8 3 **Arguments.**

9 Y shall be of type real.

10 X If Y appears, X shall be of type real with the same kind type parameter as Y. If Y has the value
 11 zero, X shall not have the value zero. If Y does not appear, X shall be of type real.

12 4 **Result Characteristics.** Same as X.

13 5 **Result Value.** If Y appears, the result is the same as the result of $\text{ATAN2D}(Y, X)$. If Y does not appear, the
 14 result has a value equal to a processor-dependent approximation to the arc tangent of X; it is expressed in
 15 degrees and lies in the range $-90 \leq \text{ATAND}(X) \leq 90$.

16 6 **Example.** $\text{ATAND}(1.0)$ has the value 45.0 (approximately).

17 16.9.26 ATANH (X)

18 1 **Description.** Inverse hyperbolic tangent function.

19 2 **Class.** [Elemental](#) function.

20 3 **Argument.** X shall be of type real or complex.

21 4 **Result Characteristics.** Same as X.

22 5 **Result Value.** The result has a value equal to a processor-dependent approximation to the inverse hyper-
 23 bolic tangent function of X. If the result is complex the imaginary part is expressed in radians and lies in
 24 the range $-\pi/2 \leq \text{AIMAG}(\text{ATANH}(X)) \leq \pi/2$.

25 6 **Example.** $\text{ATANH}(0.76159416)$ has the value 1.0 (approximately).

26 16.9.27 ATANPI (X) or ATANPI (Y, X)

27 1 **Description.** Circular arc tangent function.

28 2 **Class.** [Elemental](#) function.

29 3 **Arguments.**

30 Y shall be of type real.

31 X If Y appears, X shall be of type real with the same kind type parameter as Y. If Y has the value
 32 zero, X shall not have the value zero. If Y does not appear, X shall be of type real.

1 4 **Result Characteristics.** Same as X.

2 5 **Result Value.** If Y appears, the result is the same as the result of `ATAN2PI` (Y, X). If Y does not appear, the
3 result has a value equal to a processor-dependent approximation to the arc tangent of X; it is expressed in
4 half-revolutions and lies in the range $-0.5 \leq \text{ATANPI}(X) \leq 0.5$.

5 6 **Example.** `ATANPI` (1.0) has the value 0.25 (approximately).

6 16.9.28 `ATOMIC_ADD (ATOM, VALUE [, STAT])`

7 1 **Description.** Atomic addition.

8 2 **Class.** [Atomic subroutine](#).

9 3 **Arguments.**

10 ATOM shall be a scalar [coarray](#) or [coindexed object](#). It shall be of type integer with kind `ATOMIC-`
11 `INT_KIND` from the intrinsic module `ISO_FORTRAN_ENV`. It is an `INTENT (INOUT)` argument.
12 If an error condition occurs, ATOM becomes undefined; otherwise, it becomes defined with
13 the value of `ATOM + VALUE`.

14 VALUE shall be an integer scalar. It is an `INTENT (IN)` argument. The values of VALUE and `ATOM +`
15 VALUE shall be representable in kind `ATOMIC_INT_KIND`.

16 STAT (optional) shall be a noncoindexed integer scalar with a decimal exponent range of at least four. It is
17 an `INTENT (OUT)` argument. It is assigned a value as specified in 16.5. If an error condition
18 occurs and STAT is not present, [error termination](#) is initiated.

19 4 **Example.** `CALL ATOMIC_ADD (I [3], 42)` will cause I on [image 3](#) to become defined with the value 46 if the
20 value of I [3] is 4 when the atomic operation is executed.

21 16.9.29 `ATOMIC_AND (ATOM, VALUE [, STAT])`

22 1 **Description.** Atomic bitwise AND.

23 2 **Class.** [Atomic subroutine](#).

24 3 **Arguments.**

25 ATOM shall be a scalar [coarray](#) or [coindexed object](#). It shall be of type integer with kind `ATOMIC-`
26 `INT_KIND` from the intrinsic module `ISO_FORTRAN_ENV`. It is an `INTENT (INOUT)` argument.
27 If an error condition occurs, ATOM becomes undefined; otherwise, it becomes defined with
28 the value of `IAND (ATOM, INT (VALUE, ATOMIC_INT_KIND))`.

29 VALUE shall be an integer scalar. It is an `INTENT (IN)` argument. The value of VALUE shall be repres-
30 entable in kind `ATOMIC_INT_KIND`.

31 STAT (optional) shall be a noncoindexed integer scalar with a decimal exponent range of at least four. It is
32 an `INTENT (OUT)` argument. It is assigned a value as specified in 16.5. If an error condition
33 occurs and STAT is not present, [error termination](#) is initiated.

34 4 **Example.** `CALL ATOMIC_AND (I [3], 6)` will cause I on [image 3](#) to become defined with the value 4 if the
35 value of I [3] is 5 when the atomic operation is executed.

16.9.30 ATOMIC_CAS (ATOM, OLD, COMPARE, NEW [, STAT])

1 **Description.** Atomic compare and swap.

2 **Class.** Atomic subroutine.

3 **Arguments.**

ATOM shall be a scalar [coarray](#) or [coindexed object](#). It shall be of type integer with kind [ATOMIC_INT_KIND](#) from the intrinsic module [ISO_FORTRAN_ENV](#), or of type logical with kind [ATOMIC_LOGICAL_KIND](#) from the intrinsic module [ISO_FORTRAN_ENV](#). It is an [INTENT \(INOUT\)](#) argument. If an error condition occurs, ATOM becomes undefined; otherwise, if ATOM is of type integer and equal to COMPARE, or of type logical and equivalent to COMPARE, it becomes defined with the value of NEW.

OLD shall be scalar and of the same type and kind as ATOM. It is an [INTENT \(OUT\)](#) argument. If an error condition occurs, it becomes undefined; otherwise, it becomes defined with the value that ATOM had at the start of the atomic operation.

COMPARE shall be scalar and of the same type and kind as ATOM. It is an [INTENT \(IN\)](#) argument.

NEW shall be scalar and of the same type and kind as ATOM. It is an [INTENT \(IN\)](#) argument.

STAT (optional) shall be a noncoindexed integer scalar with a decimal exponent range of at least four. It is an [INTENT \(OUT\)](#) argument. It is assigned a value as specified in [16.5](#). If an error condition occurs and STAT is not present, [error termination](#) is initiated.

4 **Example.** If the value of I on [image 3](#) is equal to 13 at the beginning of the atomic operation performed by CALL ATOMIC_CAS (I [3], OLD, 0, 1), the value of I on [image 3](#) will be unchanged, and OLD will become defined with the value 13. If the value of I on [image 3](#) is equal to 0 at the beginning of the atomic operation performed by CALL ATOMIC_CAS (I [3], OLD, 0, 1), I on [image 3](#) will become defined with the value 1, and OLD will become defined with the value 0.

16.9.31 ATOMIC_DEFINE (ATOM, VALUE [, STAT])

1 **Description.** Define a variable atomically.

2 **Class.** Atomic subroutine.

3 **Arguments.**

ATOM shall be a scalar [coarray](#) or [coindexed object](#). It shall be of type integer with kind [ATOMIC_INT_KIND](#) from the intrinsic module [ISO_FORTRAN_ENV](#), or of type logical with kind [ATOMIC_LOGICAL_KIND](#) from the intrinsic module [ISO_FORTRAN_ENV](#). It is an [INTENT \(OUT\)](#) argument. On successful execution, it becomes defined with the value of VALUE. If an error condition occurs, it becomes undefined.

VALUE shall be scalar and of the same type as ATOM. It is an [INTENT \(IN\)](#) argument.

STAT (optional) shall be a noncoindexed integer scalar with a decimal exponent range of at least four. It is an [INTENT \(OUT\)](#) argument. It is assigned a value as specified in [16.5](#). If an error condition occurs and STAT is not present, [error termination](#) is initiated.

4 **Example.** CALL ATOMIC_DEFINE (I [3], 4) causes I on [image 3](#) to become defined with the value 4.

16.9.32 ATOMIC_FETCH_ADD (ATOM, VALUE, OLD [, STAT])

Description. Atomic fetch and add.

Class. Atomic subroutine.

Arguments.

ATOM shall be a scalar [coarray](#) or [coindexed object](#). It shall be of type integer with kind [ATOMIC_INT_KIND](#) from the intrinsic module [ISO_FORTRAN_ENV](#). It is an [INTENT \(INOUT\)](#) argument. If an error condition occurs, ATOM becomes undefined; otherwise, it becomes defined with the value of $ATOM + VALUE$.

VALUE shall be an integer scalar. It is an [INTENT \(IN\)](#) argument. The values of VALUE and $ATOM + VALUE$ shall be representable in kind [ATOMIC_INT_KIND](#).

OLD shall be scalar and of the same type and kind as ATOM. It is an [INTENT \(OUT\)](#) argument. If an error condition occurs, it becomes undefined; otherwise, it becomes defined with the value that ATOM had at the start of the atomic operation.

STAT (optional) shall be a noncoindexed integer scalar with a decimal exponent range of at least four. It is an [INTENT \(OUT\)](#) argument. It is assigned a value as specified in [16.5](#). If an error condition occurs and STAT is not present, [error termination](#) is initiated.

Example. CALL ATOMIC_FETCH_ADD (I [3], 7, J) will cause I on [image 3](#) to become defined with the value 12, and J to become defined with the value 5, if the value of I [3] is 5 when the atomic operation is executed.

16.9.33 ATOMIC_FETCH_AND (ATOM, VALUE, OLD [, STAT])

Description. Atomic fetch and bitwise AND.

Class. Atomic subroutine.

Arguments.

ATOM shall be a scalar [coarray](#) or [coindexed object](#). It shall be of type integer with kind [ATOMIC_INT_KIND](#) from the intrinsic module [ISO_FORTRAN_ENV](#). It is an [INTENT \(INOUT\)](#) argument. If an error condition occurs, ATOM becomes undefined; otherwise, it becomes defined with the value of [IAND](#) (ATOM, [INT](#) (VALUE, [ATOMIC_INT_KIND](#))).

VALUE shall be an integer scalar. It is an [INTENT \(IN\)](#) argument. The value of VALUE shall be representable in kind [ATOMIC_INT_KIND](#).

OLD shall be scalar and of the same type and kind as ATOM. It is an [INTENT \(OUT\)](#) argument. If an error condition occurs, it becomes undefined; otherwise, it becomes defined with the value that ATOM had at the start of the atomic operation.

STAT (optional) shall be a noncoindexed integer scalar with a decimal exponent range of at least four. It is an [INTENT \(OUT\)](#) argument. It is assigned a value as specified in [16.5](#). If an error condition occurs and STAT is not present, [error termination](#) is initiated.

Example. CALL ATOMIC_FETCH_AND (I [3], 6, J) will cause I on [image 3](#) to become defined with the value 4, and J to become defined with the value 5, if the value of I [3] is 5 when the atomic operation is executed.

16.9.34 ATOMIC_FETCH_OR (ATOM, VALUE, OLD [, STAT])

Description. Atomic fetch and bitwise OR.

Class. Atomic subroutine.

Arguments.

ATOM shall be a scalar [coarray](#) or [coindexed object](#). It shall be of type integer with kind [ATOMIC_INT_KIND](#) from the intrinsic module [ISO_FORTRAN_ENV](#). It is an [INTENT \(INOUT\)](#) argument. If an error condition occurs, ATOM becomes undefined; otherwise, it becomes defined with the value of [IOR](#) (ATOM, [INT](#) (VALUE, [ATOMIC_INT_KIND](#))).

VALUE shall be an integer scalar. It is an [INTENT \(IN\)](#) argument. The value of VALUE shall be representable in kind [ATOMIC_INT_KIND](#).

OLD shall be scalar and of the same type and kind as ATOM. It is an [INTENT \(OUT\)](#) argument. If an error condition occurs, it becomes undefined; otherwise, it becomes defined with the value that ATOM had at the start of the atomic operation.

STAT (optional) shall be a noncoindexed integer scalar with a decimal exponent range of at least four. It is an [INTENT \(OUT\)](#) argument. It is assigned a value as specified in [16.5](#). If an error condition occurs and STAT is not present, [error termination](#) is initiated.

Example. CALL ATOMIC_FETCH_OR (I [3], 1, J) will cause I on [image 3](#) to become defined with the value 3, and J to become defined with the value 2, if the value of I [3] is 2 when the atomic operation is executed.

16.9.35 ATOMIC_FETCH_XOR (ATOM, VALUE, OLD [, STAT])

Description. Atomic fetch and bitwise exclusive OR.

Class. Atomic subroutine.

Arguments.

ATOM shall be a scalar [coarray](#) or [coindexed object](#). It shall be of type integer with kind [ATOMIC_INT_KIND](#) from the intrinsic module [ISO_FORTRAN_ENV](#). It is an [INTENT \(INOUT\)](#) argument. If an error condition occurs, ATOM becomes undefined; otherwise, it becomes defined with the value of [IEOR](#) (ATOM, [INT](#) (VALUE, [ATOMIC_INT_KIND](#))).

VALUE shall be an integer scalar. It is an [INTENT \(IN\)](#) argument. The value of VALUE shall be representable in kind [ATOMIC_INT_KIND](#).

OLD shall be scalar and of the same type and kind as ATOM. It is an [INTENT \(OUT\)](#) argument. If an error condition occurs, it becomes undefined; otherwise, it becomes defined with the value that ATOM had at the start of the atomic operation.

STAT (optional) shall be a noncoindexed integer scalar with a decimal exponent range of at least four. It is an [INTENT \(OUT\)](#) argument. It is assigned a value as specified in [16.5](#). If an error condition occurs and STAT is not present, [error termination](#) is initiated.

Example. CALL ATOMIC_FETCH_XOR (I [3], 1, J) will cause I on [image 3](#) to become defined with the value 2, and J to become defined with the value 3, if the value of I [3] is 3 when the atomic operation is executed.

16.9.36 ATOMIC_OR (ATOM, VALUE [, STAT])

Description. Atomic bitwise OR.

Class. Atomic subroutine.

Arguments.

ATOM shall be a scalar [coarray](#) or [coindexed object](#). It shall be of type integer with kind [ATOMIC_INT_KIND](#) from the intrinsic module [ISO_FORTRAN_ENV](#). It is an [INTENT \(INOUT\)](#) argument. If an error condition occurs, ATOM becomes undefined; otherwise, it becomes defined with the value of [IOR](#) (ATOM, [INT](#) (VALUE, [ATOMIC_INT_KIND](#))).

VALUE shall be an integer scalar. It is an [INTENT \(IN\)](#) argument. The value of VALUE shall be representable in kind [ATOMIC_INT_KIND](#).

STAT (optional) shall be a noncoindexed integer scalar with a decimal exponent range of at least four. It is an [INTENT \(OUT\)](#) argument. It is assigned a value as specified in [16.5](#). If an error condition occurs and STAT is not present, [error termination](#) is initiated.

Example. CALL ATOMIC_OR (I [3], 1) will cause I on [image 3](#) to become defined with the value 3 if the value of I [3] is 2 when the atomic operation is executed.

16.9.37 ATOMIC_REF (VALUE, ATOM [, STAT])

Description. Reference a variable atomically.

Class. Atomic subroutine.

Arguments.

VALUE shall be scalar and of the same type as ATOM. It is an [INTENT \(OUT\)](#) argument. On successful execution, it becomes defined with the value of ATOM. If an error condition occurs, it becomes undefined.

ATOM shall be a scalar [coarray](#) or [coindexed object](#). It shall be of type integer with kind [ATOMIC_INT_KIND](#) from the intrinsic module [ISO_FORTRAN_ENV](#), or of type logical with kind [ATOMIC_LOGICAL_KIND](#) from the intrinsic module [ISO_FORTRAN_ENV](#). It is an [INTENT \(IN\)](#) argument.

STAT (optional) shall be a noncoindexed integer scalar with a decimal exponent range of at least four. It is an [INTENT \(OUT\)](#) argument. It is assigned a value as specified in [16.5](#). If an error condition occurs and STAT is not present, [error termination](#) is initiated.

Example. CALL ATOMIC_REF (VAL, I [3]) causes VAL to become defined with the value of I on [image 3](#).

16.9.38 ATOMIC_XOR (ATOM, VALUE [, STAT])

Description. Atomic bitwise exclusive OR.

Class. Atomic subroutine.

Arguments.

ATOM shall be a scalar [coarray](#) or [coindexed object](#). It shall be of type integer with kind [ATOMIC_INT_KIND](#) from the intrinsic module [ISO_FORTRAN_ENV](#). It is an [INTENT \(INOUT\)](#) argument. If an error condition occurs, ATOM becomes undefined; otherwise, it becomes defined with the value of [IEOR](#) (ATOM, [INT](#) (VALUE, [ATOMIC_INT_KIND](#))).

1 VALUE shall be an integer scalar. It is an **INTENT (IN)** argument. The value of VALUE shall be representable in kind **ATOMIC_INT_KIND**.

3 STAT (optional) shall be a noncoindexed integer scalar with a decimal exponent range of at least four. It is an **INTENT (OUT)** argument. It is assigned a value as specified in 16.5. If an error condition occurs and STAT is not present, **error termination** is initiated.

6 4 **Example.** CALL ATOMIC_XOR (I [3], 1) will cause I on **image 3** to become defined with the value 2 if the value of I [3] is 3 when the atomic operation is executed.

8 16.9.39 BESSEL_J0 (X)

9 1 **Description.** Bessel function of the 1st kind, order 0.

10 2 **Class.** **Elemental** function.

11 3 **Argument.** X shall be of type real.

12 4 **Result Characteristics.** Same as X.

13 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.

15 6 **Example.** BESSEL_J0 (1.0) has the value 0.765 (approximately).

16 16.9.40 BESSEL_J1 (X)

17 1 **Description.** Bessel function of the 1st kind, order 1.

18 2 **Class.** **Elemental** function.

19 3 **Argument.** X shall be of type real.

20 4 **Result Characteristics.** Same as X.

21 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.

23 6 **Example.** BESSEL_J1 (1.0) has the value 0.440 (approximately).

24 16.9.41 BESSEL_JN (N, X) or BESSEL_JN (N1, N2, X)

25 1 **Description.** Bessel functions of the 1st kind.

26 2 **Class.**

27 *Case (i):* BESSEL_JN (N,X) is an **elemental** function.

28 *Case (ii):* BESSEL_JN (N1,N2,X) is a **transformational function**.

29 3 **Arguments.**

30 N shall be of type integer and nonnegative.

31 N1 shall be an integer scalar with a nonnegative value.

32 N2 shall be an integer scalar with a nonnegative value.

33 X shall be of type real; if the function is transformational, X shall be scalar.

1 4 **Result Characteristics.** Same type and kind as X. The result of BESSEL_JN (N1, N2, X) is a rank-one array
2 with extent **MAX** (N2—N1+1, 0).

3 5 **Result Value.**

4 *Case (i):* The result value of BESSEL_JN (N, X) is a processor-dependent approximation to the Bessel
5 function of the first kind and order N of X.

6 *Case (ii):* Element *i* of the result value of BESSEL_JN (N1, N2, X) is a processor-dependent approximation
7 to the Bessel function of the first kind and order $N1+i-1$ of X.

8 6 **Example.** BESSEL_JN (2, 1.0) has the value 0.115 (approximately).

9 **16.9.42 BESSEL_Y0 (X)**

10 1 **Description.** Bessel function of the 2^{nd} kind, order 0.

11 2 **Class.** **Elemental** function.

12 3 **Argument.** X shall be of type real. Its value shall be greater than zero.

13 4 **Result Characteristics.** Same as X.

14 5 **Result Value.** The result has a value equal to a processor-dependent approximation to the Bessel function
15 of the second kind and order zero of X.

16 6 **Example.** BESSEL_Y0 (1.0) has the value 0.088 (approximately).

17 **16.9.43 BESSEL_Y1 (X)**

18 1 **Description.** Bessel function of the 2^{nd} kind, order 1.

19 2 **Class.** **Elemental** function.

20 3 **Argument.** X shall be of type real. Its value shall be greater than zero.

21 4 **Result Characteristics.** Same as X.

22 5 **Result Value.** The result has a value equal to a processor-dependent approximation to the Bessel function
23 of the second kind and order one of X.

24 6 **Example.** BESSEL_Y1 (1.0) has the value -0.781 (approximately).

25 **16.9.44 BESSEL_YN (N, X) or BESSEL_YN (N1, N2, X)**

26 1 **Description.** Bessel functions of the 2^{nd} kind.

27 2 **Class.**

28 *Case (i):* BESSEL_YN (N, X) is an **elemental** function.

29 *Case (ii):* BESSEL_YN (N1, N2, X) is a **transformational function**.

30 3 **Arguments.**

31 N shall be of type integer and nonnegative.

32 N1 shall be an integer scalar with a nonnegative value.

33 N2 shall be an integer scalar with a nonnegative value.

1 X shall be of type real; if the function is transformational, X shall be scalar. Its value shall be
2 greater than zero.

3 4 **Result Characteristics.** Same type and kind as X. The result of BESSEL_YN (N1, N2, X) is a rank-one array
4 with extent [MAX](#) (N2—N1+1, 0).

5 5 **Result Value.**

6 *Case (i):* The result value of BESSEL_YN (N, X) is a processor-dependent approximation to the Bessel
7 function of the second kind and order N of X.

8 *Case (ii):* Element *i* of the result value of BESSEL_YN (N1, N2, X) is a processor-dependent approxima-
9 tion to the Bessel function of the second kind and order N1+*i* — 1 of X.

10 6 **Example.** BESSEL_YN (2, 1.0) has the value —1.651 (approximately).

11 16.9.45 BGE (I, J)

12 1 **Description.** Bitwise greater than or equal to.

13 2 **Class.** [Elemental](#) function.

14 3 **Arguments.**

15 I shall be of type integer or a [boz-literal-constant](#).

16 J shall be of type integer or a [boz-literal-constant](#).

17 4 **Result Characteristics.** Default logical.

18 5 **Result Value.** The result is true if the sequence of bits represented by I is greater than or equal to the
19 sequence of bits represented by J, according to the method of bit sequence comparison in [16.3.2](#); otherwise
20 the result is false.

21 6 The interpretation of a [boz-literal-constant](#) as a sequence of bits is described in [7.7](#). The interpretation of
22 an integer value as a sequence of bits is described in [16.3](#).

23 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
24 the value false.

25 16.9.46 BGT (I, J)

26 1 **Description.** Bitwise greater than.

27 2 **Class.** [Elemental](#) function.

28 3 **Arguments.**

29 I shall be of type integer or a [boz-literal-constant](#).

30 J shall be of type integer or a [boz-literal-constant](#).

31 4 **Result Characteristics.** Default logical.

32 5 **Result Value.** The result is true if the sequence of bits represented by I is greater than the sequence of
33 bits represented by J, according to the method of bit sequence comparison in [16.3.2](#); otherwise the result
34 is false.

1 6 The interpretation of a *boz-literal-constant* as a sequence of bits is described in 7.7. The interpretation of
2 an integer value as a sequence of bits is described in 16.3.

3 7 **Example.** BGT (Z'FF', Z'FC') has the value true. BGT (0, −1) has the value false.

4 16.9.47 BIT_SIZE (I)

5 1 **Description.** Number of bits in integer model 16.3.

6 2 **Class.** *Inquiry function*.

7 3 **Argument.** I shall be of type integer. It may be a scalar or an array.

8 4 **Result Characteristics.** Scalar integer with the same kind type parameter as I.

9 5 **Result Value.** The result has the value of the number of bits z of the model integer defined for bit manipu-
10 lation contexts in 16.3.

11 6 **Example.** BIT_SIZE (1) has the value 32 if z of the model is 32.

12 16.9.48 BLE (I, J)

13 1 **Description.** Bitwise less than or equal to.

14 2 **Class.** *Elemental function*.

15 3 **Arguments.**

16 I shall be of type integer or a *boz-literal-constant*.

17 J shall be of type integer or a *boz-literal-constant*.

18 4 **Result Characteristics.** Default logical.

19 5 **Result Value.** The result is true if the sequence of bits represented by I is less than or equal to the sequence
20 of bits represented by J, according to the method of bit sequence comparison in 16.3.2; otherwise the result
21 is false.

22 6 The interpretation of a *boz-literal-constant* as a sequence of bits is described in 7.7. The interpretation of
23 an integer value as a sequence of bits is described in 16.3.

24 7 **Example.** BLE (0, J) has the value true for any value of J. BLE (−1, 0) has the value false.

25 16.9.49 BLT (I, J)

26 1 **Description.** Bitwise less than.

27 2 **Class.** *Elemental function*.

28 3 **Arguments.**

29 I shall be of type integer or a *boz-literal-constant*.

30 J shall be of type integer or a *boz-literal-constant*.

31 4 **Result Characteristics.** Default logical.

32 5 **Result Value.** The result is true if the sequence of bits represented by I is less than the sequence of bits

1 represented by J, according to the method of bit sequence comparison in 16.3.2; otherwise the result is
2 false.

3 6 The interpretation of a *boz-literal-constant* as a sequence of bits is described in 7.7. The interpretation of
4 an integer value as a sequence of bits is described in 16.3.

5 7 **Example.** BLT (0, −1) has the value true. BLT (Z'FF', Z'FC') has the value false.

6 16.9.50 BTEST (I, POS)

7 1 **Description.** Test single bit in an integer.

8 2 **Class.** *Elemental* function.

9 3 **Arguments.**

10 I shall be of type integer.

11 POS shall be of type integer. It shall be nonnegative and be less than BIT_SIZE (I).

12 4 **Result Characteristics.** Default logical.

13 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
14 of I has the value 0. The model for the interpretation of an integer value as a sequence of bits is in 16.3.

15 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
16 $\begin{bmatrix} \text{false} & \text{false} \\ \text{false} & \text{true} \end{bmatrix}$ and the value of BTEST (2, A) is $\begin{bmatrix} \text{true} & \text{false} \\ \text{false} & \text{false} \end{bmatrix}$.

17 16.9.51 CEILING (A [, KIND])

18 1 **Description.** Least integer greater than or equal to A.

19 2 **Class.** *Elemental* function.

20 3 **Arguments.**

21 A shall be of type real.

22 KIND (optional) shall be a scalar integer *constant expression*.

23 4 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the value
24 of KIND; otherwise, the kind type parameter is that of default integer type.

25 5 **Result Value.** The result has a value equal to the least integer greater than or equal to A.

26 6 **Examples.** CEILING (3.7) has the value 4. CEILING (−3.7) has the value −3.

27 16.9.52 CHAR (I [, KIND])

28 1 **Description.** Character from code value.

29 2 **Class.** *Elemental* function.

30 3 **Arguments.**

1 I shall be of type integer with a value in the range $0 \leq I \leq n - 1$, where n is the number of
2 characters in the [collating sequence](#) associated with the specified kind type parameter.

3 KIND (optional) shall be a scalar integer [constant expression](#).

4 4 **Result Characteristics.** Character of length one. If KIND is present, the kind type parameter is that spe-
5 cified by the value of KIND; otherwise, the kind type parameter is that of default character.

6 5 **Result Value.** The result is the character in position I of the [collating sequence](#) associated with the spe-
7 cified kind type parameter. ICHAR (CHAR (I, KIND (C))) shall have the value I for $0 \leq I \leq n - 1$ and
8 CHAR (ICHAR (C), KIND (C)) shall have the value C for any character C capable of representation in the
9 processor.

10 6 **Example.** CHAR (88) has the value 'X' on a processor using the [ASCII collating sequence](#) for default char-
11 acters.

12 16.9.53 CMPLX (X [, KIND]) or CMPLX (X [, Y, KIND])

13 1 **Description.** Conversion to complex type.

14 2 **Class.** [Elemental](#) function.

15 3 **Arguments for CMPLX(X [, KIND]).**

16 X shall be of type complex.

17 KIND (optional) shall be a scalar integer [constant expression](#).

18 4 **Arguments for CMPLX(X [, Y, KIND]).**

19 X shall be of type integer or real, or a [boz-literal-constant](#).

20 Y (optional) shall be of type integer or real, or a [boz-literal-constant](#).

21 KIND (optional) shall be a scalar integer [constant expression](#).

22 5 **Result Characteristics.** The result is of type complex. If KIND is present, the kind type parameter is that
23 specified by the value of KIND; otherwise, the kind type parameter is that of default real kind.

24 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
25 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
26 of CMPLX ([REAL](#) (X), [AIMAG](#) (X), KIND). The result of CMPLX (X, Y, KIND) has the complex value whose real
27 part is [REAL](#) (X, KIND) and whose imaginary part is [REAL](#) (Y, KIND).

28 7 **Example.** CMPLX (−3) has the value (−3.0, 0.0).

29 16.9.54 CO_BROADCAST (A, SOURCE_IMAGE [, STAT, ERRMSG])

30 1 **Description.** Broadcast value to images.

31 2 **Class.** [Collective subroutine](#).

32 3 **Arguments.**

33 A shall have the same shape, type, and type parameter values, in corresponding references. It
34 shall not be [polymorphic](#) or a [coindexed object](#). It is an [INTENT \(INOUT\)](#) argument. If no error
35 condition occurs, A becomes defined, as if by [intrinsic assignment](#), on all images in the [current](#)
36 [team](#) with the value of A on image SOURCE_IMAGE, including (re)allocation of any allocatable

potential subobject component, and setting the dynamic type of any polymorphic allocatable potential subobject component.

SOURCE_IMAGE shall be an integer scalar. It is an **INTENT (IN)** argument. Its value shall be that of an image index of an image in the current team. The value shall be the same in all corresponding references.

STAT (optional) shall be a noncoindexed integer scalar with a decimal exponent range of at least four. It is an **INTENT (OUT)** argument.

ERRMSG (optional) shall be a noncoindexed default character scalar. It is an **INTENT (INOUT)** argument.

4 The semantics of STAT and ERRMSG are described in 16.6.

5 **Example.** If A is the array [1, 5, 3] on image one, after execution of

```
CALL CO_BROADCAST (A, 1)
```

the value of A on all images of the current team is [1, 5, 3].

16.9.55 CO_MAX (A [, RESULT_IMAGE, STAT, ERRMSG])

1 **Description.** Compute maximum value across images.

2 **Class.** Collective subroutine.

3 **Arguments.**

A shall be of type integer, real, or character. It shall have the same shape, type, and type parameter values, in corresponding references. It shall not be a **coindexed object**. It is an **INTENT (INOUT)** argument. If it is scalar, the computed value is equal to the maximum value of A in all corresponding references. If it is an array each element of the computed value is equal to the maximum value of all corresponding elements of A in all corresponding references. The computed value is assigned to A if no error condition occurs, and either RESULT_IMAGE is absent, or the executing image is the one identified by RESULT_IMAGE. Otherwise, A becomes undefined.

RESULT_IMAGE (optional) shall be an integer scalar. It is an **INTENT (IN)** argument. Its presence, and value if present, shall be the same in all corresponding references. If it is present, its value shall be that of an image index in the current team.

STAT (optional) shall be a noncoindexed integer scalar with a decimal exponent range of at least four. It is an **INTENT (OUT)** argument.

ERRMSG (optional) shall be a noncoindexed default character scalar. It is an **INTENT (INOUT)** argument.

4 The semantics of STAT and ERRMSG are described in 16.6.

5 **Example.** If the number of images in the current team is two and A is the array [1, 5, 3] on one image and [4, 1, 6] on the other image, the value of A after executing the statement CALL CO_MAX (A) is [4, 5, 6] on both images.

16.9.56 CO_MIN (A [, RESULT_IMAGE, STAT, ERRMSG])

1 **Description.** Compute minimum value across images.

2 **Class.** Collective subroutine.

3 **Arguments.**

A shall be of type integer, real, or character. It shall have the same shape, type, and type parameter values, in corresponding references. It shall not be a [coindexed object](#). It is an [INTENT \(INOUT\)](#) argument. If it is scalar, the computed value is equal to the minimum value of A in all corresponding references. If it is an array each element of the computed value is equal to the minimum value of all corresponding elements of A in all corresponding references.

The computed value is assigned to A if no error condition occurs, and either RESULT_IMAGE is absent, or the executing image is the one identified by RESULT_IMAGE. Otherwise, A becomes undefined.

RESULT_IMAGE (optional) shall be an integer scalar. It is an [INTENT \(IN\)](#) argument. Its presence, and value if present, shall be the same in all corresponding references. If it is present, its value shall be that of an image index in the current team.

STAT (optional) shall be a noncoindexed integer scalar with a decimal exponent range of at least four. It is an [INTENT \(OUT\)](#) argument.

ERRMSG (optional) shall be a noncoindexed default character scalar. It is an [INTENT \(INOUT\)](#) argument.

4 The semantics of STAT and ERRMSG are described in [16.6](#).

5 **Example.** If the number of images in the current team is two and A is the array [1, 5, 3] on one image and [4, 1, 6] on the other image, the value of A after executing the statement CALL CO_MIN (A) is [1, 1, 3] on both images.

16.9.57 CO_REDUCE (A, OPERATION [, RESULT_IMAGE, STAT, ERRMSG])

1 **Description.** Generalized reduction across images.

2 **Class.** [Collective subroutine](#).

3 **Arguments.**

A shall not be polymorphic. It shall not be of a type with an [ultimate component](#) that is allocatable or a pointer. It shall have the same shape, type, and type parameter values, in corresponding references. It shall not be a [coindexed object](#). It is an [INTENT \(INOUT\)](#) argument. If A is scalar, the computed value is the result of the reduction operation of applying OPERATION to the values of A in all corresponding references. If A is an array, each element of the computed value is equal to the result of the reduction operation of applying OPERATION to corresponding elements of A in all corresponding references.

The computed value is assigned to A if no error condition occurs, and either RESULT_IMAGE is absent, or the executing image is the one identified by RESULT_IMAGE. Otherwise, A becomes undefined.

OPERATION shall be a pure function with exactly two arguments; the result and each argument shall be a scalar, nonallocatable, noncoarray, nonpointer, nonpolymorphic data object with the same type and type parameters as A. The arguments shall not be optional. If one argument has the [ASYNCHRONOUS](#), [TARGET](#), or [VALUE](#) attribute, the other shall have that attribute. OPERATION shall implement a mathematically associative operation. OPERATION shall be the same function on all images in corresponding references.

The computed value of a reduction operation over a set of values is the result of an iterative process. Each iteration involves the evaluation of OPERATION (x, y) for x and y in the set, the removal of x and y from the set, and the addition of the value of OPERATION (x, y) to the set. The process terminates when the set has only one element; this is the computed value.

RESULT_IMAGE (optional) shall be an integer scalar. It is an **INTENT (IN)** argument. Its presence, and value if present, shall be the same in all corresponding references. If it is present, its value shall be that of an image index in the current team.

STAT (optional) shall be a noncoindexed integer scalar with a decimal exponent range of at least four. It is an **INTENT (OUT)** argument.

ERRMSG (optional) shall be a noncoindexed default character scalar. It is an **INTENT (INOUT)** argument.

4 The semantics of STAT and ERRMSG are described in 16.6.

5 **Example.** The subroutine below demonstrates how to use CO_REDUCE to create a collective counterpart to the intrinsic function ALL:

```

10      SUBROUTINE co_all(boolean)
11          LOGICAL, INTENT(INOUT) :: boolean
12          CALL CO_REDUCE(boolean,both)
13      CONTAINS
14          PURE FUNCTION both(lhs,rhs) RESULT(lhs_and_rhs)
15              LOGICAL, INTENT(IN) :: lhs,rhs
16              LOGICAL :: lhs_and_rhs
17              lhs_and_rhs = lhs .AND. rhs
18          END FUNCTION both
19      END SUBROUTINE co_all

```

NOTE

If the OPERATION function is not mathematically commutative, the result of calling CO_REDUCE can depend on the order of evaluations.

16.9.58 CO_SUM (A [, RESULT_IMAGE, STAT, ERRMSG])

1 **Description.** Compute sum across images.

2 **Class.** Collective subroutine.

3 **Arguments.**

A shall be of numeric type. It shall have the same shape, type, and type parameter values, in corresponding references. It shall not be a **coindexed object**. It is an **INTENT (INOUT)** argument. If it is scalar, the computed value is equal to a processor-dependent approximation to the sum of the values of A in corresponding references. If it is an array, each element of the computed value is equal to a processor-dependent approximation to the sum of all corresponding elements of A in corresponding references.

The computed value is assigned to A if no error condition occurs, and either RESULT_IMAGE is absent, or the executing image is the one identified by RESULT_IMAGE. Otherwise, A becomes undefined.

RESULT_IMAGE (optional) shall be an integer scalar. It is an **INTENT (IN)** argument. Its presence, and value if present, shall be the same in all corresponding references. If it is present, its value shall be that of an image index in the current team.

STAT (optional) shall be a noncoindexed integer scalar with a decimal exponent range of at least four. It is an **INTENT (OUT)** argument.

ERRMSG (optional) shall be a noncoindexed default character scalar. It is an **INTENT (INOUT)** argument.

4 The semantics of STAT and ERRMSG are described in 16.6.

1 5 **Example.** If the number of images in the current team is two and A is the array [1, 5, 3] on one image and
 2 [4, 1, 6] on the other image, the value of A after executing the statement CALL CO_SUM(A) is [5, 6, 9] on
 3 both images.

4 16.9.59 COMMAND_ARGUMENT_COUNT ()

5 1 **Description.** Number of command arguments.

6 2 **Class.** [Transformational function](#).

7 3 **Argument.** None.

8 4 **Result Characteristics.** Default integer scalar.

9 5 **Result Value.** The result value is equal to the number of command arguments available. If there are no
 10 command arguments available or if the processor does not support command arguments, then the result
 11 has the value zero. If the processor has a concept of a command name, the command name does not count
 12 as one of the command arguments.

13 6 **Example.** See [16.9.93](#).

14 16.9.60 CONJG (Z)

15 1 **Description.** Conjugate of a complex number.

16 2 **Class.** [Elemental function](#).

17 3 **Argument.** Z shall be of type complex.

18 4 **Result Characteristics.** Same as Z.

19 5 **Result Value.** If Z has the value (x, y) , the result has the value $(x, -y)$.

20 6 **Example.** CONJG ((2.0, 3.0)) has the value (2.0, -3.0).

21 16.9.61 COS (X)

22 1 **Description.** Cosine function.

23 2 **Class.** [Elemental function](#).

24 3 **Argument.** X shall be of type real or complex.

25 4 **Result Characteristics.** Same as X.

26 5 **Result Value.** The result has a value equal to a processor-dependent approximation to $\cos(X)$. If X is of
 27 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
 28 radians.

29 6 **Example.** COS (1.0) has the value 0.54030231 (approximately).

30 16.9.62 COSD (X)

31 1 **Description.** Degree cosine function.

32 2 **Class.** [Elemental function](#).

1 3 **Argument.** X shall be of type real.

2 4 **Result Characteristics.** Same as X.

3 5 **Result Value.** The result has a value equal to a processor-dependent approximation to the cosine of X,
4 which is regarded as a value in degrees.

5 6 **Example.** COSD (180.0) has the value -1.0 (approximately).

6 16.9.63 COSH (X)

7 1 **Description.** Hyperbolic cosine function.

8 2 **Class.** [Elemental](#) function.

9 3 **Argument.** X shall be of type real or complex.

10 4 **Result Characteristics.** Same as X.

11 5 **Result Value.** The result has a value equal to a processor-dependent approximation to cosh(X). If X is of
12 type complex its imaginary part is regarded as a value in radians.

13 6 **Example.** COSH (1.0) has the value 1.5430806 (approximately).

14 16.9.64 COSHAPE (COARRAY [, KIND])

15 1 **Description.** Sizes of [codimensions](#) of a [coarray](#).

16 2 **Class.** [Inquiry](#) function.

17 3 **Arguments.**

18 COARRAY shall be a [coarray](#) of any type. It shall not be an unallocated [allocatable coarray](#). If its *desig-*
19 *nator* has more than one *part-ref*, the rightmost *part-ref* shall have nonzero [corank](#).

20 KIND (optional) shall be a scalar integer [constant expression](#).

21 4 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the value
22 of KIND; otherwise the kind type parameter is that of default integer type. The result is an array of [rank](#)
23 one whose size is equal to the [corank](#) of COARRAY.

24 5 **Result Value.** The result has a value whose i^{th} element is equal to the size of the i^{th} [codimension](#) of COAR-
25 RAY, as given by $UCOBOUND(COARRAY, i) - LCOBOUND(COARRAY, i) + 1$.

26 6 **Example.**

27 The following code allocates the coarray D with the same size in each codimension as that of the coarray C,
28 with the lower cobound 1.

```
29 REAL, ALLOCATABLE :: C[:, :], D[:, :]
30 INTEGER, ALLOCATABLE :: COSHAPE_C(:)
31 ...
32 COSHAPE_C = COSHAPE(C)
33 ALLOCATE ( D[COSHAPE_C(1), *] )
```

16.9.65 COSPI (X)

Description. Circular cosine function.

Class. [Elemental](#) function.

Argument. X shall be of type real.

Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to the cosine of X, which is regarded as a value in half-revolutions; thus COSPI (X) is approximately equal to [COS](#) ($X \times \pi$).

Example. COSPI (1.0) has the value -1.0 (approximately).

16.9.66 COUNT (MASK [, DIM, KIND])

Description. Array reduced by counting true values.

Class. [Transformational](#) function.

Arguments.

MASK shall be a logical array.

DIM (optional) shall be an integer scalar with a value in the range $1 \leq \text{DIM} \leq n$, where n is the [rank](#) of MASK. The corresponding [actual argument](#) shall not be an optional dummy argument, a disassociated pointer, or an unallocated allocatable.

KIND (optional) shall be a scalar integer [constant expression](#).

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type. The result is scalar if DIM is absent or $n = 1$; otherwise, the result has [rank](#) $n - 1$ and shape $[d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n]$ where $[d_1, d_2, \dots, d_n]$ is the shape of MASK.

Result Value.

Case (i): If DIM is absent or MASK has rank one, the result has a value equal to the number of true elements of MASK or has the value zero if MASK has size zero.

Case (ii): If DIM is present and MASK has rank $n > 1$, the value of element $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$ of the result is equal to the number of true elements of MASK $(s_1, s_2, \dots, s_{\text{DIM}-1}, \cdot, s_{\text{DIM}+1}, \dots, s_n)$.

Examples.

Case (i): The value of COUNT ([.TRUE.](#), [.FALSE.](#), [.TRUE.](#)) is 2.

Case (ii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ and C is the array $\begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}$, COUNT (B /= C, DIM = 1) is [2, 0, 1] and COUNT (B /= C, DIM = 2) is [1, 2].

16.9.67 CPU_TIME (TIME)

Description. Processor time used.

Class. Subroutine.

1 3 **Argument.** TIME shall be a real scalar. It is an **INTENT (OUT)** argument. If the processor cannot provide a
 2 meaningful value for the time, it is assigned a processor-dependent negative value; otherwise, it is assigned
 3 a processor-dependent approximation to the processor time in seconds. Whether the value assigned is an
 4 approximation to the amount of time used by the invoking **image**, or the amount of time used by the whole
 5 program, is processor dependent.

6 4 **Example.**

```
7      REAL T1, T2
8      ...
9      CALL CPU_TIME(T1)
10     ... Code to be timed.
11     CALL CPU_TIME(T2)
12     WRITE (*,*) 'Time taken by code was ', T2-T1, ' seconds'
```

13 writes the processor time taken by a piece of code.

NOTE

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.

14 16.9.68 CSHIFT (ARRAY, SHIFT [, DIM])

15 1 **Description.** Circular shift of an array.

16 2 **Class.** Transformational function.

17 3 **Arguments.**

18 ARRAY may be of any type. It shall be an array.

19 SHIFT shall be of type integer and shall be scalar if ARRAY has **rank** one; otherwise, it shall be scalar
 20 or of **rank** $n - 1$ and of shape $[d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n]$ where $[d_1, d_2, \dots, d_n]$ is the shape
 21 of ARRAY.

22 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
 23 ARRAY. If DIM is absent, it is as if it were present with the value 1.

24 4 **Result Characteristics.** The result is of the type and type parameters of ARRAY, and has the shape of
 25 ARRAY.

26 5 **Result Value.**

27 Case (i): If ARRAY has **rank** one, element i of the result is $\text{ARRAY}(\text{LBOUND}(\text{ARRAY}, 1) + \text{MODULO}(i +$
 28 $\text{SHIFT} - 1, \text{SIZE}(\text{ARRAY})))$.

29 Case (ii): If ARRAY has **rank** greater than one, section $(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)$ of the result has
 30 a value equal to $\text{CSHIFT}(\text{ARRAY}(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n), sh, 1)$, where sh is SHIFT or
 31 SHIFT $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$.

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 $\text{CSHIFT}(V, \text{SHIFT} = 2)$ which has the value $[3, 4, 5, 6, 1, 2]$; $\text{CSHIFT}(V, \text{SHIFT} = -2)$ 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 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 $\text{CSHIFT}(M, \text{SHIFT} = -1, \text{DIM} = 2)$ is $\begin{bmatrix} 3 & 1 & 2 \\ 6 & 4 & 5 \\ 9 & 7 & 8 \end{bmatrix}$,

and the value of $\text{CSHIFT}(M, \text{SHIFT} = [-1, 1, 0], \text{DIM} = 2)$ is $\begin{bmatrix} 3 & 1 & 2 \\ 5 & 6 & 4 \\ 7 & 8 & 9 \end{bmatrix}$.

16.9.69 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 $YYYYMMDD$, where $YYYY$ is the year in the Gregorian calendar, MM is the month within the year, and DD is the day within the month. The characters of this value shall all be decimal digits. If there is no date available, DATE is assigned all blanks.

TIME (optional) shall be a default character scalar. It is an [INTENT \(OUT\)](#) argument. It is assigned a value of the form $hhmmss.sss$, where hh is the hour of the day, mm is the minutes of the hour, and $ss.sss$ is the seconds and milliseconds of the minute. Except for the decimal point, the characters of this value shall all be decimal digits. If there is no clock available, TIME is assigned all blanks.

ZONE (optional) shall be a default character scalar. It is an [INTENT \(OUT\)](#) argument. It is assigned a value of the form $+hhmm$ or $-hhmm$, where hh and mm are the time difference with respect to Coordinated Universal Time (UTC) in hours and minutes, respectively. The characters of this value following the sign character shall all be decimal digits. If this information is not available, ZONE is assigned all blanks.

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 $-\text{HUGE}(\text{VALUES})$ if there is no date available;

VALUES (2) the month of the year, or $-\text{HUGE}(\text{VALUES})$ if there is no date available;

VALUES (3) the day of the month, or $-\text{HUGE}(\text{VALUES})$ if there is no date available;

VALUES (4) the time difference from Coordinated Universal Time (UTC) in minutes, or $-\text{HUGE}(\text{VALUES})$ if this information is not available;

VALUES (5) the hour of the day, in the range of 0 to 23, or $-\text{HUGE}(\text{VALUES})$ if there is no clock;

VALUES (6) the minutes of the hour, in the range 0 to 59, or $-\text{HUGE}(\text{VALUES})$ if there is no clock;

VALUES (7) the seconds of the minute, in the range 0 to 60, or $-\text{HUGE}(\text{VALUES})$ if there is no clock;

VALUES (8) the milliseconds of the second, in the range 0 to 999, or $-\text{HUGE}(\text{VALUES})$ 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.

Example. If run in Geneva, Switzerland on April 12, 2008 at 15:27:35.5 with a system configured for the local time zone, this example would have assigned the value 20080412 to BIG_BEN (1), the value 152735.500 to BIG_BEN (2), the value +0100 to BIG_BEN (3), and the value [2008, 4, 12, 60, 15, 27, 35, 500] to DATE_TIME.

```
INTEGER DATE_TIME (8)
CHARACTER (LEN = 10) BIG_BEN (3)
CALL DATE_AND_TIME (BIG_BEN (1), BIG_BEN (2), BIG_BEN (3), DATE_TIME)
```

NOTE

These forms are compatible with the representations defined in ISO 8601:2004. UTC is established by the International Bureau of Weights and Measures (BIPM, i.e. Bureau International des Poids et Mesures) and the International Earth Rotation Service (IERS).

16.9.70 DBLE (A)

Description. Conversion to double precision real.

Class. [Elemental](#) function.

Argument. A shall be of type integer, real, complex, or a [boz-literal-constant](#).

Result Characteristics. Double precision real.

Result Value. The result has the value REAL (A, KIND (0.0D0)).

Example. DBLE (−3) has the value −3.0D0.

16.9.71 DIGITS (X)

Description. Significant digits in numeric model.

Class. [Inquiry function](#).

Argument. X shall be of type integer or real. It may be a scalar or an array.

Result Characteristics. Default integer scalar.

Result Value. The result has the value q if X is of type integer and p if X is of type real, where q and p are as defined in [16.4](#) for the model representing numbers of the same type and kind type parameter as X.

Example. DIGITS (X) has the value 24 for real X whose model is as in [16.4](#), [NOTE](#).

16.9.72 DIM (X, Y)

Description. Maximum of $X - Y$ and zero.

Class. [Elemental](#) function.

Arguments.

X shall be of type integer or real.

1 Y shall be of the same type and kind type parameter as X.

2 4 **Result Characteristics.** Same as X.

3 5 **Result Value.** The value of the result is the maximum of $X - Y$ and zero.

4 6 **Example.** DIM (−3.0, 2.0) has the value 0.0.

5 16.9.73 DOT_PRODUCT (VECTOR_A, VECTOR_B)

6 1 **Description.** Dot product of two vectors.

7 2 **Class.** Transformational function.

8 3 **Arguments.**

9 VECTOR_A shall be of **numeric type** (integer, real, or complex) or of logical type. It shall be a rank-one
10 array.

11 VECTOR_B shall be of **numeric type** if VECTOR_A is of **numeric type** or of type logical if VECTOR_A is of
12 type logical. It shall be a rank-one array. It shall be of the same size as VECTOR_A.

13 4 **Result Characteristics.** If the arguments are of **numeric type**, the type and **kind type parameter** of the res-
14 ult are those of the expression VECTOR_A * VECTOR_B determined by the types and kinds of the arguments
15 according to 10.1.9.3. If the arguments are of type logical, the result is of type logical with the kind type
16 parameter of the expression VECTOR_A .AND. VECTOR_B according to 10.1.9.3. The result is scalar.

17 5 **Result Value.**

18 *Case (i):* If VECTOR_A is of type integer or real, the result has the value SUM (VECTOR_A*VECTOR_B).
19 If the vectors have size zero, the result has the value zero.

20 *Case (ii):* If VECTOR_A is of type complex, the result has the value SUM (CONJG (VECTOR_A)*VECTOR_-
21 B). If the vectors have size zero, the result has the value zero.

22 *Case (iii):* If VECTOR_A is of type logical, the result has the value ANY (VECTOR_A .AND. VECTOR_B). If
23 the vectors have size zero, the result has the value false.

24 6 **Example.** DOT_PRODUCT ([1, 2, 3], [2, 3, 4]) has the value 20.

25 16.9.74 DPROD (X, Y)

26 1 **Description.** Double precision real product.

27 2 **Class.** Elemental function.

28 3 **Arguments.**

29 X shall be default real.

30 Y shall be default real.

31 4 **Result Characteristics.** Double precision real.

32 5 **Result Value.** The result has a value equal to a processor-dependent approximation to the product of X
33 and Y. DPROD (X, Y) should have the same value as DBLE (X) * DBLE (Y).

34 6 **Example.** DPROD (−3.0, 2.0) has the value −6.0D0.

16.9.75 DSHIFTL (I, J, SHIFT)

Description. Combined left shift.

Class. [Elemental](#) function.

Arguments.

I shall be of type integer or a [boz-literal-constant](#).

J shall be of type integer or a [boz-literal-constant](#). If both I and J are of type integer, they shall have the same kind type parameter. I and J shall not both be [boz-literal-constants](#).

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).

Result Characteristics. Same as I if I is of type integer; otherwise, same as J.

Result Value. If either I or J is a [boz-literal-constant](#), it is first converted as if by the intrinsic function [INT](#) to type integer with the kind type parameter of the other. The rightmost SHIFT bits of the result value are the same as the leftmost bits of J, and the remaining bits of the result value are the same as the rightmost bits of I. This is equal to IOR (SHIFTL (I, SHIFT), SHIFTR (J, BIT_SIZE (J)–SHIFT)). The model for the interpretation of an integer value as a sequence of bits is in [16.3](#).

Examples. DSHIFTL (1, 2**30, 2) has the value 5 if default integer has 32 bits. DSHIFTL (I, I, SHIFT) has the same result value as ISHFTC (I, SHIFT).

16.9.76 DSHIFTR (I, J, SHIFT)

Description. Combined right shift.

Class. [Elemental](#) function.

Arguments.

I shall be of type integer or a [boz-literal-constant](#).

J shall be of type integer or a [boz-literal-constant](#). If both I and J are of type integer, they shall have the same kind type parameter. I and J shall not both be [boz-literal-constants](#).

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).

Result Characteristics. Same as I if I is of type integer; otherwise, same as J.

Result Value. If either I or J is a [boz-literal-constant](#), it is first converted as if by the intrinsic function [INT](#) to type integer with the kind type parameter of the other. The leftmost SHIFT bits of the result value are the same as the rightmost bits of I, and the remaining bits of the result value are the same as the leftmost bits of J. This is equal to IOR (SHIFTL (I, BIT_SIZE (I)–SHIFT), SHIFTR (J, SHIFT)). The model for the interpretation of an integer value as a sequence of bits is in [16.3](#).

Examples. DSHIFTR (1, 16, 3) has the value $2^{29} + 2$ if default integer has 32 bits. DSHIFTR (I, I, SHIFT) has the same result value as ISHFTC (I, –SHIFT).

16.9.77 EOSHIFT (ARRAY, SHIFT [, BOUNDARY, DIM])

Description. End-off shift of the elements of an array.

Class. Transformational function.

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 $[d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n]$ where $[d_1, d_2, \dots, d_n]$ 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 $[d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n]$. BOUNDARY is permitted to be absent only for the types in Table 16.4, and in this case it is as if it were present with the scalar value shown, converted if necessary to the kind type parameter value of ARRAY.

Table 16.4 — Default BOUNDARY values for EOSHIFT

Type of ARRAY	Value of BOUNDARY
Integer	0
Real	0.0
Complex	(0.0, 0.0)
Logical	.FALSE.
Character (<i>len</i>)	<i>len</i> blanks

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. If DIM is absent, it is as if it were present with the value 1.

Result Characteristics. The result has the type, type parameters, and shape of ARRAY.

Result Value. Element $(s_1, s_2, \dots, s_n)$ of the result has the value $\text{ARRAY}(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}} + sh, s_{\text{DIM}+1}, \dots, s_n)$ where sh is SHIFT or $\text{SHIFT}(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$ provided the inequality $\text{LBOUND}(\text{ARRAY}, \text{DIM}) \leq s_{\text{DIM}} + sh \leq \text{UBOUND}(\text{ARRAY}, \text{DIM})$ holds and is otherwise BOUNDARY or $\text{BOUNDARY}(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$.

Examples.

Case (i): If V is the array [1, 2, 3, 4, 5, 6], the effect of shifting V end-off to the left by 3 positions is achieved by $\text{EOSHIFT}(V, \text{SHIFT} = 3)$, which has the value [4, 5, 6, 0, 0, 0]; $\text{EOSHIFT}(V, \text{SHIFT} = -2, \text{BOUNDARY} = 99)$ achieves an end-off shift to the right by 2 positions with the boundary value of 99 and has the value [99, 99, 1, 2, 3, 4].

Case (ii): The rows of an array of **rank** two may all be shifted by the same amount or by different amounts

and the boundary elements can be the same or different. If M is the array $\begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix}$, then

the value of $\text{EOSHIFT}(M, \text{SHIFT} = -1, \text{BOUNDARY} = '*', \text{DIM} = 2)$ is $\begin{bmatrix} * & A & B \\ * & D & E \\ * & G & H \end{bmatrix}$, and the value

of $\text{EOSHIFT}(M, \text{SHIFT} = [-1, 1, 0], \text{BOUNDARY} = ['*', '/', '?'], \text{DIM} = 2)$ is $\begin{bmatrix} * & A & B \\ E & F & / \\ G & H & I \end{bmatrix}$.

16.9.78 EPSILON (X)

Description. Model number that is small compared to 1.

Class. [Inquiry function](#).

Argument. X shall be of type real. It may be a scalar or an array.

Result Characteristics. Scalar of the same type and kind type parameter as X.

Result Value. The result has the value b^{1-p} where b and p are as defined in [16.4](#) for the model representing numbers of the same type and kind type parameter as X.

Example. EPSILON (X) has the value 2^{-23} for real X whose model is as in [16.4](#), [NOTE](#).

16.9.79 ERF (X)

Description. Error function.

Class. [Elemental function](#).

Argument. X shall be of type real.

Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to the error function of X, $\frac{2}{\sqrt{\pi}} \int_0^X \exp(-t^2) dt$.

Example. ERF (1.0) has the value 0.843 (approximately).

16.9.80 ERFC (X)

Description. Complementary error function.

Class. [Elemental function](#).

Argument. X shall be of type real.

Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to the complementary error function of X, $1 - \text{ERF}(X)$; this is equivalent to $\frac{2}{\sqrt{\pi}} \int_X^\infty \exp(-t^2) dt$.

Example. ERFC (1.0) has the value 0.157 (approximately).

16.9.81 ERFC_SCALED (X)

Description. Scaled complementary error function.

Class. [Elemental function](#).

Argument. X shall be of type real.

Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to the exponentially-scaled complementary error function of X, $\exp(X^2) \frac{2}{\sqrt{\pi}} \int_X^\infty \exp(-t^2) dt$.

6 **Example.** ERFC_SCALED (20.0) has the value 0.02817434874 (approximately).

NOTE

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 60559:2020 single precision arithmetic. The exponentially-scaled complementary error function is asymptotic to $1/(X\sqrt{\pi})$. As such it does not underflow until $X > \text{HUGE}(X)/\sqrt{\pi}$.

16.9.82 EVENT_QUERY (EVENT, COUNT [, STAT])

1 **Description.** Query event count.

2 **Class.** Subroutine.

3 **Arguments.**

EVENT shall be an [event variable](#) (16.10.2.10). It shall not be [coindexed](#). It is an [INTENT \(IN\)](#) argument. The EVENT argument is accessed atomically with respect to the execution of [EVENT POST statements](#) in unordered segments, in exact analogy to [atomic subroutines](#).

COUNT shall be an integer scalar with a decimal exponent range no smaller than that of default integer. It is an [INTENT \(OUT\)](#) argument. If no error condition occurs, COUNT is assigned the value of the count of EVENT; otherwise, it is assigned the value -1 .

STAT (optional) shall be a noncoindexed integer scalar with a decimal exponent range of at least four. It is an [INTENT \(OUT\)](#) argument. If the STAT argument is present, it is assigned a processor-dependent positive value if an error condition occurs; otherwise it is assigned the value zero. If the STAT argument is not present and an error condition occurs, [error termination](#) is initiated.

4 **Example.** If EVENT is an [event variable](#) for which there have been no successful posts or waits in preceding [segments](#), and for which there are no posts or waits in an unordered [segment](#), after execution of

```
CALL EVENT_QUERY (EVENT, COUNT)
```

the integer variable COUNT will have the value zero. If there have been ten successful posts to EVENT and two successful waits without an [UNTIL_COUNT= specifier](#) in preceding [segments](#), and for which there are no posts or waits in an unordered [segment](#), after execution of

```
CALL EVENT_QUERY (EVENT, COUNT)
```

the variable COUNT will have the value eight.

NOTE

Execution of EVENT_QUERY does not imply any synchronization.

16.9.83 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.

16.9.84 EXP (X)

1 **Description.** Exponential function.

2 **Class.** **Elemental** function.

3 **Argument.** X shall be of type real or complex.

4 **Result Characteristics.** Same as X.

5 **Result Value.** The result has a value equal to a processor-dependent approximation to e^X . If X is of type complex, its imaginary part is regarded as a value in radians.

6 **Example.** EXP (1.0) has the value 2.7182818 (approximately).

16.9.85 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 (16.4), provided X is nonzero and e is within the range for default integers. If X has the value zero, the result has the value zero. If X is an IEEE infinity or NaN, the result has the value **HUGE** (0).

Examples. EXPONENT (1.0) has the value 1 and EXPONENT (4.1) has the value 3 for reals whose model is as in 16.4, NOTE.

16.9.86 EXTENDS_TYPE_OF (A, MOLD)

Description. Dynamic type extension inquiry.

Class. Inquiry function.

Arguments.

A shall be an object of **extensible declared** type or **unlimited polymorphic**. If it is a **polymorphic** pointer, it shall not have an undefined association status.

MOLD shall be an object of **extensible declared** type or **unlimited polymorphic**. If it is a **polymorphic** pointer, it shall not have an undefined association status.

Result Characteristics. Default logical scalar.

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 1

The **dynamic type** of a **disassociated** pointer or unallocated **allocatable** variable is its **declared type**.

NOTE 2

The test performed by EXTENDS_TYPE_OF is not the same as the test performed by the type guard **CLASS IS**. The test performed by EXTENDS_TYPE_OF does not consider **kind type parameters**.

Example. Given the declarations and assignments

```
TYPE T1
  REAL C
END TYPE
TYPE, EXTENDS (T1) :: T2
END TYPE
CLASS(T1), POINTER :: P, Q
ALLOCATE (P)
ALLOCATE (T2 :: Q)
```

the result of EXTENDS_TYPE_OF (P, Q) will be false, and the result of EXTENDS_TYPE_OF (Q, P) will be true.

16.9.87 FAILED_IMAGES ([TEAM, KIND])

Description. Indices of failed images.

Class. Transformational function.

Arguments.

TEAM (optional) shall be a scalar of type **TEAM_TYPE** from the intrinsic module **ISO_FORTRAN_ENV**. Its value shall be that of the **current** or an **ancestor** team. If TEAM is absent, the team specified is the **current team**.

1 KIND (optional) shall be a scalar integer [constant expression](#).

2 4 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the value
3 of KIND; otherwise, the kind type parameter is that of default integer type. The result is an array of rank
4 one whose size is equal to the number of images in the specified team that are known by the invoking image
5 to have failed.

6 5 **Result Value.** The elements of the result are the values of the image indices of the known failed images
7 in the specified team, in numerically increasing order. If the executing image has previously executed an
8 [image control statement](#) whose STAT= specifier assigned the value [STAT_FAILED_IMAGE](#) from the intrinsic
9 module [ISO_FORTRAN_ENV](#), or referenced a [collective subroutine](#) whose STAT argument was set to [STAT_-](#)
10 [FAILED_IMAGE](#), at least one image in the set of images participating in that [image control statement](#) or
11 [collective subroutine](#) reference shall be known to have failed.

12 6 **Examples.** If image 3 is the only image in the [current team](#) that is known by the invoking image to have
13 failed, [FAILED_IMAGES\(\)](#) will have the value [3]. If there are no images in the current team that are known
14 by the invoking image to have failed, the value of [FAILED_IMAGES\(\)](#) will be a zero-sized array.

15 16.9.88 FINDLOC (ARRAY, VALUE, DIM [, MASK, KIND, BACK]) or 16 FINDLOC (ARRAY, VALUE [, MASK, KIND, BACK])

16 1 **Description.** Location(s) of a specified value.

17 2 **Class.** [Transformational function](#).

18 3 **Arguments.**

19 ARRAY shall be an array of intrinsic type.

20 VALUE shall be scalar and in type conformance with ARRAY, as specified in Table 10.2 for the operator
21 == or the operator [.EQV.](#)

22 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.

23 MASK (optional) shall be of type logical and shall be [conformable](#) with ARRAY.

24 KIND (optional) shall be a scalar integer [constant expression](#).

25 BACK (optional) shall be a logical scalar.

26 4 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the value of
27 KIND; otherwise the kind type parameter is that of default integer type. If DIM does not appear, the result
28 is an array of [rank](#) one and of size equal to the [rank](#) of ARRAY; otherwise, the result is of [rank](#) $n - 1$ and
29 shape $[d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n]$, where $[d_1, d_2, \dots, d_n]$ is the shape of ARRAY.

30 5 **Result Value.**

31 *Case (i):* The result of [FINDLOC \(ARRAY, VALUE\)](#) is a rank-one array whose element values are the val-
32 ues of the subscripts of an element of ARRAY whose value matches VALUE. If there is such a
33 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
34 of ARRAY. If no elements match VALUE or ARRAY has size zero, all elements of the result are
35 zero.

36 *Case (ii):* The result of [FINDLOC \(ARRAY, VALUE, MASK = MASK\)](#) is a rank-one array whose element
37 values are the values of the subscripts of an element of ARRAY, corresponding to a true element
38 of MASK, whose value matches VALUE. If there is such a value, the i^{th} element value is in the
39 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 $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$ of the result is equal to FINDLOC (ARRAY $(s_1, s_2, \dots, s_{\text{DIM}-1}, \vdots, s_{\text{DIM}+1}, \dots, s_n)$, VALUE, DIM=1 [, MASK = MASK $(s_1, s_2, \dots, s_{\text{DIM}-1}, \vdots, s_{\text{DIM}+1}, \dots, s_n)$])).

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.

16.9.89 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.

16.9.90 FRACTION (X)

Description. Fractional part of number.

Class. [Elemental](#) function.

Argument. X shall be of type real.

Result Characteristics. Same as X.

Result Value. The result has the value $X \times b^{-e}$, where b and e are as defined in [16.4](#) for the representation of X in the extended real model for the kind of X. If X has the value zero, the result is zero. If X is an [IEEE NaN](#), the result is that [NaN](#). If X is an IEEE infinity, the result is an [IEEE NaN](#).

Example. FRACTION (3.0) has the value 0.75 for reals whose model is as in [16.4](#), [NOTE](#).

16.9.91 GAMMA (X)

Description. Gamma function.

Class. [Elemental](#) function.

Argument. X shall be of type real. Its value shall not be a negative integer or zero.

Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to the gamma function of X,

$$\Gamma(X) = \begin{cases} \int_0^\infty t^{X-1} \exp(-t) dt & X > 0 \\ \int_0^\infty t^{X-1} \left(\exp(-t) - \sum_{k=0}^n \frac{(-t)^k}{k!} \right) dt & -n-1 < X < -n, n \text{ an integer } \geq 0 \end{cases}$$

Example. GAMMA (1.0) has the value 1.000 (approximately).

16.9.92 GET_COMMAND ([COMMAND, LENGTH, STATUS, ERRMSG])

Description. Get program invocation command.

Class. Subroutine.

Arguments.

COMMAND (optional) shall be a default character scalar. It is an [INTENT \(OUT\)](#) argument. It is assigned the entire command by which the program was invoked. If the command cannot be determined, COMMAND is assigned all blanks.

LENGTH (optional) shall be a scalar of type integer with a decimal exponent range of at least four. It is an [INTENT \(OUT\)](#) argument. It is assigned the significant length of the command by which the program was invoked. The significant length may include trailing blanks if the processor allows commands with significant trailing blanks. This length does not consider any possible truncation or padding in assigning the command to the COMMAND argument; in fact the COMMAND argument need not even be present. If the command length cannot be determined, a length of 0 is assigned.

STATUS (optional) shall be a scalar of type integer with a decimal exponent range of at least four. It is an **INTENT (OUT)** argument. It is assigned the value -1 if the COMMAND argument is present and has a length less than the significant length of the command. It is assigned a processor-dependent positive value if the command retrieval fails. Otherwise it is assigned the value 0.

ERRMSG (optional) shall be a default character scalar. It is an **INTENT (INOUT)** argument. It is assigned a processor-dependent explanatory message if the command retrieval fails. Otherwise, it is unchanged.

- 4 **Example.** If the program below is invoked with the command “example” on a processor that supports command retrieval, it will display “Hello example”.

```
PROGRAM hello
  CHARACTER(:), ALLOCATABLE :: cmd
  CALL GET_COMMAND(cmd)
  PRINT *, 'Hello ', cmd
END PROGRAM
```

16.9.93 GET_COMMAND_ARGUMENT (NUMBER [, VALUE, LENGTH, STATUS, ERRMSG])

- 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.

ERRMSG (optional) shall be a default character scalar. It is an **INTENT (INOUT)** argument. It is assigned a processor-dependent explanatory message if the optional argument STATUS is, or would be if present, assigned a positive value. Otherwise, it is unchanged.

- 4 **Example.** On a processor that supports command arguments, the following program displays the arguments of the command by which it was invoked.

```

PROGRAM show_arguments
  INTEGER :: i
  CHARACTER :: command*32, arg*128
  CALL get_command_argument(0, command)
  WRITE (*,*) "Command name is: ", command
  DO i = 1, command_argument_count()
    CALL get_command_argument(i, arg)
    WRITE (*,*) "Argument ", i, " is ", arg
  END DO
END PROGRAM show_arguments

```

16.9.94 GET_ENVIRONMENT_VARIABLE (NAME [, VALUE, LENGTH, STATUS, TRIM_NAME, ERRMSG])

- 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 assigned the value of its length. Otherwise LENGTH is assigned the value zero.

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 assigned the value zero. STATUS is assigned the value -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.

ERRMSG (optional) shall be a default character scalar. It is an **INTENT (INOUT)** argument. It is assigned a processor-dependent explanatory message if the optional argument STATUS is, or would be if present, assigned a positive value. Otherwise, it is unchanged.

- 4 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.

1 5 **Example.** If the value of the environment variable DATAFILE is datafile.dat, executing the statement sequence below will assign the value 'datafile.dat' to FILENAME.

```
3      CHARACTER (:),ALLOCATABLE :: FILENAME
4      CALL GET_ENVIRONMENT_VARIABLE("DATAFILE", FILENAME)
```

5 16.9.95 GET_TEAM ([LEVEL])

6 1 **Description.** Team.

7 2 **Class.** Transformational function.

8 3 **Argument.** LEVEL (optional) shall be a scalar integer whose value is equal to one of the [named constants](#) INITIAL_TEAM, PARENT_TEAM, or CURRENT_TEAM from the intrinsic module ISO_FORTRAN_ENV.

10 4 **Result Characteristics.** Scalar of type TEAM_TYPE from the intrinsic module ISO_FORTRAN_ENV.

11 5 **Result Value.** The result is a TEAM_TYPE value that identifies the [current team](#) if LEVEL is not present, present with the value CURRENT_TEAM, or if the [current team](#) is the [initial team](#). Otherwise, the result identifies the [parent team](#) if LEVEL is present with the value PARENT_TEAM, and identifies the [initial team](#) if LEVEL is present with the value INITIAL_TEAM.

15 6 **Examples.**

```
16      PROGRAM EXAMPLE1
17          USE,INTRINSIC :: ISO_FORTRAN_ENV, ONLY: TEAM_TYPE
18          TYPE(TEAM_TYPE) :: WORLD_TEAM, TEAM2
19
20          ! Define a team variable representing the initial team
21          WORLD_TEAM = GET_TEAM()
22      END PROGRAM
23
24      SUBROUTINE EXAMPLE2 (A)
25          USE,INTRINSIC :: ISO_FORTRAN_ENV, ONLY: TEAM_TYPE
26          REAL A[*]
27          TYPE(TEAM_TYPE) :: NEW_TEAM, PARENT_TEAM
28
29          ... ! Form NEW_TEAM
30
31          PARENT_TEAM = GET_TEAM ()
32
33          CHANGE TEAM (NEW_TEAM)
34
35          ! Reference image 1 in parent's team
36          A [1,TEAM=PARENT_TEAM] = 4.2
37
38          ! Reference image 1 in current team
39          A [1] = 9.0
40      END TEAM
41      END SUBROUTINE EXAMPLE2
```

NOTE

Because the result of GET_TEAM is of type TEAM_TYPE from the intrinsic module ISO_FORTRAN_ENV, a [program unit](#) that assigns the result of a reference to GET_TEAM to a local variable will also need access to the definition of TEAM_TYPE from the intrinsic module ISO_FORTRAN_ENV.

16.9.96 HUGE (X)

Description. Largest model value or last enumeration value.

Class. [Inquiry function](#).

Argument. X shall be of type integer or real, or of enumeration type. It may be a scalar or an array.

Result Characteristics. Scalar of the same type and kind type parameter as X.

Result Value. The result has the value $r^q - 1$ if X is of type integer and $(1 - b^{-p})b^{e_{\max}}$ if X is of type real, where r , q , b , p , and $e_{\max}$ are as defined in [16.4](#) for the model representing numbers of the same type and kind type parameter as X. If X is of enumeration type, the result has the value of the last enumerator in the type definition.

Example. HUGE (X) has the value $(1 - 2^{-24}) \times 2^{127}$ for real X whose model is as in [16.4](#), [NOTE](#).

16.9.97 HYPOT (X, Y)

Description. Euclidean distance function.

Class. [Elemental function](#).

Arguments.

X shall be of type real.

Y shall be of type real with the same kind type parameter as X.

Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to the Euclidean distance, $\sqrt{X^2 + Y^2}$, without undue overflow or underflow.

Example. HYPOT (3.0, 4.0) has the value 5.0 (approximately).

16.9.98 IACHAR (C [, KIND])

Description. ASCII code value for character.

Class. [Elemental function](#).

Arguments.

C shall be of type character and of length one.

KIND (optional) shall be a scalar integer [constant expression](#).

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default integer type.

Result Value. If C is in the [collating sequence](#) defined by the codes specified in ISO/IEC 646:1991 (International Reference Version), the result is the position of C in that sequence; it is nonnegative and less than or equal to 127. The value of the result is processor dependent if C is not in the [ASCII collating sequence](#). The results are consistent with the LGE, LGT, LLE, and LLT comparison functions. For example, if LLE (C, D) is true, IACHAR (C) <= IACHAR (D) is true where C and D are any two characters representable by the processor.

1 6 **Example.** IACHAR ('X') has the value 88.

2 16.9.99 IALL (ARRAY, DIM [, MASK]) or IALL (ARRAY [, MASK])

3 1 **Description.** Array reduced by IAND function.

4 2 **Class.** Transformational function.

5 3 **Arguments.**

6 ARRAY shall be an array of type integer.

7 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.

8 MASK (optional) shall be of type logical and shall be **conformable** with ARRAY.

9 4 **Result Characteristics.** The result is of the same type and kind type parameter as ARRAY. It is scalar if
10 DIM does not appear or if ARRAY has **rank** one; otherwise, the result is an array of **rank** $n - 1$ and shape
11 $[d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n]$ where $[d_1, d_2, \dots, d_n]$ is the shape of ARRAY.

12 5 **Result Value.**

13 *Case (i):* If ARRAY has size zero the result value is equal to NOT (INT (0, KIND (ARRAY))). Otherwise,
14 the result of IALL (ARRAY) has a value equal to the bitwise AND of all the elements of ARRAY.

15 *Case (ii):* The result of IALL (ARRAY, MASK=MASK) has a value equal to IALL (PACK (ARRAY, MASK)).

16 *Case (iii):* The result of IALL (ARRAY, DIM=DIM [, MASK=MASK]) has a value equal to that of IALL (AR-
17 RAY [, MASK=MASK]) if ARRAY has **rank** one. Otherwise, the value of element $(s_1, s_2, \dots,$
18 $s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$ of the result is equal to IALL (ARRAY $(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)$
19 $[, \text{MASK} = \text{MASK} (s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)])$.

20 6 **Examples.** IALL ([14, 13, 11]) has the value 8. IALL ([14, 13, 11], MASK=[.true., .false., .true.]) has the value
21 10.

22 16.9.100 IAND (I, J)

23 1 **Description.** Bitwise AND.

24 2 **Class.** Elemental function.

25 3 **Arguments.**

26 I shall be of type integer or a **boz-literal-constant**.

27 J shall be of type integer or a **boz-literal-constant**. If both I and J are of type integer, they shall
28 have the same kind type parameter. I and J shall not both be **boz-literal-constants**.

29 4 **Result Characteristics.** Same as I if I is of type integer; otherwise, same as J.

30 5 **Result Value.** If either I or J is a **boz-literal-constant**, it is first converted as if by the intrinsic function INT
31 to type integer with the kind type parameter of the other. The result has the value obtained by combining
32 I and J bit-by-bit according to Table 16.5.

Table 16.5 — Bitwise result table for IAND

I	J	IAND (I, J)
1	1	1
1	0	0

Bitwise result table for IAND (cont.)

I	J	IAND (I, J)
0	1	0
0	0	0

6 The model for the interpretation of an integer value as a sequence of bits is in 16.3.

7 **Example.** IAND (1, 3) has the value 1.

16.9.101 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 $[d_1, d_2, \dots, d_{DIM-1}, d_{DIM+1}, \dots, d_n]$ where $[d_1, d_2, \dots, d_n]$ 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 $(s_1, s_2, \dots, s_{DIM-1}, s_{DIM+1}, \dots, s_n)$ of the result is equal to IANY (ARRAY $(s_1, s_2, \dots, s_{DIM-1}, :, s_{DIM+1}, \dots, s_n)$ [, MASK = MASK $(s_1, s_2, \dots, s_{DIM-1}, :, s_{DIM+1}, \dots, s_n)$])).

6 **Examples.** IANY ([14, 13, 8]) has the value 15. IANY ([14, 13, 8], MASK=[.true., .false., .true.]) has the value 14.

16.9.102 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.

1 5 **Result Value.** The result has the value of the sequence of bits of I, except that bit POS is zero. The model
2 for the interpretation of an integer value as a sequence of bits is in 16.3.

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)
4 is [29, 27, 23, 15].

5 16.9.103 IBITS (I, POS, LEN)

6 1 **Description.** Specified sequence of bits.

7 2 **Class.** Elemental function.

8 3 **Arguments.**

9 I shall be of type integer.

10 POS shall be of type integer. It shall be nonnegative and POS + LEN shall be less than or equal to
11 BIT_SIZE (I).

12 LEN shall be of type integer and nonnegative.

13 4 **Result Characteristics.** Same as I.

14 5 **Result Value.** The result has the value of the sequence of LEN bits in I beginning at bit POS, right-adjusted
15 and with all other bits zero. The model for the interpretation of an integer value as a sequence of bits is in
16 16.3.

17 6 **Example.** IBITS (14, 1, 3) has the value 7.

18 16.9.104 IBSET (I, POS)

19 1 **Description.** I with bit POS replaced by one.

20 2 **Class.** Elemental function.

21 3 **Arguments.**

22 I shall be of type integer.

23 POS shall be of type integer. It shall be nonnegative and less than BIT_SIZE (I).

24 4 **Result Characteristics.** Same as I.

25 5 **Result Value.** The result has the value of the sequence of bits of I, except that bit POS is one. The model for
26 the interpretation of an integer value as a sequence of bits is in 16.3.

27 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)
28 is [2, 4, 8, 16].

29 16.9.105 ICHAR (C [, KIND])

30 1 **Description.** Code value for character.

31 2 **Class.** Elemental function.

32 3 **Arguments.**

33 C shall be of type character and of length one. Its value shall be that of a character capable of
34 representation in the processor.

- 1 KIND (optional) shall be a scalar integer [constant expression](#).
- 2 4 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the value
3 of KIND; otherwise, the kind type parameter is that of default integer type.
- 4 5 **Result Value.** The result is the position of C in the processor [collating sequence](#) associated with the kind
5 type parameter of C; it is nonnegative and less than n , where n is the number of characters in the [collating](#)
6 [sequence](#). The [kind type parameter](#) of the result shall specify an integer kind that is capable of represent-
7 ing n . For any characters C and D capable of representation in the processor, $C \leq D$ is true if and only if
8 $\text{ICHAR}(C) \leq \text{ICHAR}(D)$ is true and $C = D$ is true if and only if $\text{ICHAR}(C) = \text{ICHAR}(D)$ is true.
- 9 6 **Example.** ICHAR ('X') has the value 88 on a processor using the [ASCII collating sequence](#) for default char-
10 acters.

11 16.9.106 IEOR (I, J)

- 12 1 **Description.** Bitwise exclusive OR.
- 13 2 **Class.** [Elemental](#) function.
- 14 3 **Arguments.**
- 15 I shall be of type integer or a [boz-literal-constant](#).
- 16 J shall be of type integer or a [boz-literal-constant](#). If both I and J are of type integer, they shall
17 have the same kind type parameter. I and J shall not both be [boz-literal-constants](#).
- 18 4 **Result Characteristics.** Same as I if I is of type integer; otherwise, same as J.
- 19 5 **Result Value.** If either I or J is a [boz-literal-constant](#), it is first converted as if by the intrinsic function [INT](#)
20 to type integer with the kind type parameter of the other. The result has the value obtained by combining
21 I and J bit-by-bit according to Table 16.6.

Table 16.6 — Bitwise result table for IEOR

I	J	IEOR (I, J)
1	1	0
1	0	1
0	1	1
0	0	0

- 22 6 The model for the interpretation of an integer value as a sequence of bits is in [16.3](#).
- 23 7 **Example.** IEOR (1, 3) has the value 2.

24 16.9.107 IMAGE_INDEX (COARRAY, SUB) or (COARRAY, SUB, TEAM) or (COARRAY, SUB, TEAM_NUMBER)

- 25 1 **Description.** [Image index](#) from [cosubscripts](#).
- 26 2 **Class.** [Transformational](#) function.
- 27 3 **Arguments.**
- 28 COARRAY shall be a [coarray](#) of any type. If its [designator](#) has more than one [part-ref](#), the rightmost
29 [part-ref](#) shall have nonzero [corank](#). If TEAM_NUMBER appears and the [current team](#) is not

the **initial team**, it shall be **established** in the **parent** of the **current team**. If TEAM_NUMBER appears and the **current team** is the **initial team**, it shall be **established** in the **initial team** and the value of TEAM_NUMBER shall be the **team number** for the **initial team**. If TEAM appears, it shall be **established** in that **team**. If neither TEAM nor TEAM_NUMBER appears, it shall be **established** in the **current team**.

SUB shall be a rank-one integer array of size equal to the **corank** of COARRAY.

TEAM shall be a scalar of type **TEAM_TYPE** from the intrinsic module **ISO_FORTRAN_ENV**, with a value that identifies the **current** or an **ancestor team**.

TEAM_NUMBER shall be an integer scalar. It shall identify the **initial team** or a **sibling team** of the **current team**.

4 **Result Characteristics.** Default integer scalar.

5 **Result Value.** If the value of SUB is a valid sequence of **cosubscripts** for COARRAY in the **team** specified by TEAM or TEAM_NUMBER, or the **current team** if neither TEAM nor TEAM_NUMBER appears, the result is the **index** of the corresponding **image** in that **team**. Otherwise, the result is zero.

6 **Examples.** If A and B are declared as A [0:*] and B (10, 20) [10, 0:9, 0:*] respectively, IMAGE_INDEX (A, [0]) has the value 1 and IMAGE_INDEX (B, [3, 1, 2]) has the value 213 (on any **image**, provided the number of **images** is at least 213).

16.9.108 IMAGE_STATUS (IMAGE [, TEAM])

1 **Description.** Image execution state.

2 **Class.** **Elemental** function.

3 **Arguments.**

IMAGE shall be of type integer. Its value shall be positive and less than or equal to the number of **images** in the specified **team**.

TEAM (optional) shall be a scalar of type **TEAM_TYPE** from the intrinsic module **ISO_FORTRAN_ENV**. Its value shall represent the **current** or an **ancestor** team. If TEAM is absent, the team specified is the **current team**.

4 **Result Characteristics.** Default integer.

5 **Result Value.** The result value is **STAT_FAILED_IMAGE** from the intrinsic module **ISO_FORTRAN_ENV** if the specified image has failed, **STAT_STOPPED_IMAGE** from the intrinsic module **ISO_FORTRAN_ENV** if that image has initiated **normal termination**, and zero otherwise.

6 **Example.** If image 3 of the current team has failed, IMAGE_STATUS (3) has the value STAT_FAILED_IMAGE.

16.9.109 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.

1 BACK (optional) shall be of type logical.

2 KIND (optional) shall be a scalar integer [constant expression](#).

3 4 **Result Characteristics.** Integer. If KIND is present, the [kind type parameter](#) is that specified by the value
4 of KIND; otherwise the [kind type parameter](#) is that of default integer type.

5 5 **Result Value.**

6 *Case (i):* If $\text{STRING \% LEN} < \text{SUBSTRING \% LEN}$, the result has the value zero.

7 *Case (ii):* Otherwise, if there is an integer I in the range $1 \leq I \leq \text{STRING \% LEN} - \text{SUBSTRING \% LEN} +$
8 1, such that $\text{STRING}(I : I + \text{SUBSTRING \% LEN} - 1)$ is equal to SUBSTRING, the result has the
9 value of the smallest such I if BACK is absent or present with the value false, and the greatest
10 such I if BACK is present with the value true.

11 *Case (iii):* Otherwise, the result has the value zero.

12 6 **Examples.** INDEX ('FORTRAN', 'R') has the value 3.

13 INDEX ('FORTRAN', 'R', BACK = [.TRUE.](#)) has the value 5.

14 16.9.110 INT (A [, KIND])

15 1 **Description.** Conversion to integer type.

16 2 **Class.** [Elemental](#) function.

17 3 **Arguments.**

18 A shall be of type integer, real, or complex, or of enum or enumeration type, or a [boz-literal-](#)
19 [constant](#).

20 KIND (optional) shall be a scalar integer [constant expression](#).

21 4 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the value
22 of KIND; otherwise, the kind type parameter is that of default integer type.

23 5 **Result Value.**

24 *Case (i):* If A is of type integer, $\text{INT}(A) = A$.

25 *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)
26 is the integer whose magnitude is the largest integer that does not exceed the magnitude of A
27 and whose sign is the same as the sign of A.

28 *Case (iii):* If A is of type complex, $\text{INT}(A) = \text{INT}(\text{REAL}(A, \text{KIND}(A)))$.

29 *Case (iv):* If A is of enum type, INT (A) has the value of the corresponding integer value.

30 *Case (v):* If A is of enumeration type, INT (A) has the value of the ordinal position of A.

31 *Case (vi):* If A is a [boz-literal-constant](#), the value of the result is the value whose bit sequence according
32 to the model in [16.3](#) is the same as that of A as modified by padding or truncation according
33 to [16.3.3](#). The interpretation of a bit sequence whose most significant bit is 1 is processor
34 dependent.

35 6 **Example.** INT (−3.7) has the value −3.

16.9.111 IOR (I, J)

Description. Bitwise inclusive OR.

Class. [Elemental](#) function.

Arguments.

I shall be of type integer or a [boz-literal-constant](#).

J shall be of type integer or a [boz-literal-constant](#). If both I and J are of type integer, they shall have the same kind type parameter. I and J shall not both be [boz-literal-constants](#).

Result Characteristics. Same as I if I is of type integer; otherwise, same as J.

Result Value. If either I or J is a [boz-literal-constant](#), it is first converted as if by the intrinsic function [INT](#) to type integer with the kind type parameter of the other. The result has the value obtained by combining I and J bit-by-bit according to Table [16.7](#).

Table 16.7 — Bitwise result table for IOR

I	J	IOR (I, J)
1	1	1
1	0	1
0	1	1
0	0	0

The model for the interpretation of an integer value as a sequence of bits is in [16.3](#).

Example. IOR (5, 3) has the value 7.

16.9.112 IPARITY (ARRAY, DIM [, MASK]) or IPARITY (ARRAY [, MASK])

Description. Array reduced by IEOR function.

Class. [Transformational](#) function.

Arguments.

ARRAY shall be of type integer. It shall be an array.

DIM shall be an integer scalar with a value in the range $1 \leq \text{DIM} \leq n$, where n is the [rank](#) of ARRAY.

MASK (optional) shall be of type logical and shall be [conformable](#) with ARRAY.

Result Characteristics. The result is of the same type and kind type parameter as ARRAY. It is scalar if DIM does not appear; otherwise, the result has [rank](#) $n - 1$ and shape $[d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n]$ where $[d_1, d_2, \dots, d_n]$ is the shape of ARRAY.

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 $(s_1, s_2,$

1 $\dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n$) of the result is equal to IPARITY (ARRAY ($s_1, s_2, \dots, s_{\text{DIM}-1}, \vdots, s_{\text{DIM}+1}, \dots,$
2 s_n) [, MASK = MASK ($s_1, s_2, \dots, s_{\text{DIM}-1}, \vdots, s_{\text{DIM}+1}, \dots, s_n$))]).

3 6 **Examples.** IPARITY ([14, 13, 8]) has the value 11. IPARITY ([14, 13, 8], MASK=[.true., .false., .true.]) has
4 the value 6.

5 16.9.113 ISHFT (I, SHIFT)

6 1 **Description.** Logical shift.

7 2 **Class.** [Elemental](#) function.

8 3 **Arguments.**

9 I shall be of type integer.

10 SHIFT shall be of type integer. The absolute value of SHIFT shall be less than or equal to BIT_SIZE (I).

11 4 **Result Characteristics.** Same as I.

12 5 **Result Value.** The result has the value obtained by shifting the bits of I by SHIFT positions. If SHIFT is
13 positive, the shift is to the left; if SHIFT is negative, the shift is to the right; if SHIFT is zero, no shift is
14 performed. Bits shifted out from the left or from the right, as appropriate, are lost. Zeros are shifted in
15 from the opposite end. The model for the interpretation of an integer value as a sequence of bits is in [16.3](#).

16 6 **Example.** ISHFT (3, 1) has the value 6.

17 16.9.114 ISHFTC (I, SHIFT [, SIZE])

18 1 **Description.** Circular shift of the rightmost bits.

19 2 **Class.** [Elemental](#) function.

20 3 **Arguments.**

21 I shall be of type integer.

22 SHIFT shall be of type integer. The absolute value of SHIFT shall be less than or equal to SIZE.

23 SIZE (optional) shall be of type integer. The value of SIZE shall be positive and shall not exceed BIT_SIZE (I).

24 If SIZE is absent, it is as if it were present with the value of BIT_SIZE (I).

25 4 **Result Characteristics.** Same as I.

26 5 **Result Value.** The result has the value obtained by shifting the SIZE rightmost bits of I circularly by SHIFT
27 positions. If SHIFT is positive, the shift is to the left; if SHIFT is negative, the shift is to the right; and if
28 SHIFT is zero, no shift is performed. No bits are lost. The unshifted bits are unaltered. The model for the
29 interpretation of an integer value as a sequence of bits is in [16.3](#).

30 6 **Example.** ISHFTC (3, 2, 3) has the value 5.

31 16.9.115 IS_CONTIGUOUS (ARRAY)

32 1 **Description.** Array contiguity test ([8.5.7](#)).

33 2 **Class.** [Inquiry function](#).

1 3 **Argument.** ARRAY may be of any type. It shall be [assumed-rank](#) or an array. If it is a pointer it shall be
2 associated.

3 4 **Result Characteristics.** Default logical scalar.

4 5 **Result Value.** The result has the value true if ARRAY has rank zero or is [contiguous](#), and false otherwise.

5 6 **Example.** After the [pointer assignment](#) AP => TARGET (1:10:2), IS_CONTIGUOUS (AP) has the value false.

6 16.9.116 IS_IOSTAT_END (I)

7 1 **Description.** IOSTAT value test for end of file.

8 2 **Class.** [Elemental](#) function.

9 3 **Argument.** I shall be of type integer.

10 4 **Result Characteristics.** Default logical.

11 5 **Result Value.** The result has the value true if and only if I is a value for the [stat-variable](#) in an [IOSTAT=](#)
12 [specifier](#) (12.11.5) that would indicate an end-of-file condition.

13 16.9.117 IS_IOSTAT_EOR (I)

14 1 **Description.** IOSTAT value test for end of record.

15 2 **Class.** [Elemental](#) function.

16 3 **Argument.** I shall be of type integer.

17 4 **Result Characteristics.** Default logical.

18 5 **Result Value.** The result has the value true if and only if I is a value for the [stat-variable](#) in an [IOSTAT=](#)
19 [specifier](#) (12.11.5) that would indicate an end-of-record condition.

20 16.9.118 KIND (X)

21 1 **Description.** Value of the kind type parameter of X.

22 2 **Class.** [Inquiry function](#).

23 3 **Argument.** X may be of any intrinsic type. It may be a scalar or an array.

24 4 **Result Characteristics.** Default integer scalar.

25 5 **Result Value.** The result has a value equal to the kind type parameter value of X.

26 6 **Example.** KIND (0.0) has the kind type parameter value of default real.

27 16.9.119 LBOUND (ARRAY [, DIM, KIND])

28 1 **Description.** Lower bound(s).

29 2 **Class.** [Inquiry function](#).

30 3 **Arguments.**

1 ARRAY shall be **assumed-rank** or an array. It shall not be an unallocated **allocatable** variable or a
2 pointer that is not associated.

3 DIM (optional) shall be an integer scalar with a value in the range $1 \leq \text{DIM} \leq n$, where n is the **rank**
4 of ARRAY. The corresponding **actual argument** shall not be an optional dummy argument, a
5 disassociated pointer, or an unallocated allocatable.

6 KIND (optional) shall be a scalar integer **constant expression**.

7 4 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the value
8 of KIND; otherwise the kind type parameter is that of default integer type. The result is scalar if DIM is
9 present; otherwise, the result is an array of **rank** one and size n , where n is the **rank** of ARRAY.

10 5 **Result Value.**

11 *Case (i):* If DIM is present, ARRAY is a **whole array**, and either ARRAY is an **assumed-size array** of **rank**
12 DIM or dimension DIM of ARRAY has nonzero extent, the result has a value equal to the lower
13 bound for subscript DIM of ARRAY. Otherwise, if DIM is present, the result value is 1.

14 *Case (ii):* LBOUND (ARRAY) has a value whose i^{th} element is equal to LBOUND (ARRAY, i), for $i = 1, 2,$
15 $\dots, n$, where n is the **rank** of ARRAY. LBOUND (ARRAY, KIND=KIND) has a value whose i^{th}
16 element is equal to LBOUND (ARRAY, i , KIND), for $i = 1, 2, \dots, n$, where n is the **rank** of
17 ARRAY.

NOTE

If ARRAY is **assumed-rank** and has rank zero, DIM cannot be present since it cannot satisfy the requirement $1 \leq \text{DIM} \leq 0$.

18 6 **Examples.** If A is declared by the statement

19 `REAL A (2:3, 7:10)`

20 then LBOUND (A) is [2, 7] and LBOUND (A, DIM=2) is 7.

21 16.9.120 LCOBOUND (COARRAY [, DIM, KIND])

22 1 **Description.** Lower **cobound**(s) of a **coarray**.

23 2 **Class.** **Inquiry function**.

24 3 **Arguments.**

25 COARRAY shall be a **coarray** and may be of any type. It may be a scalar or an array. If it is **allocatable**
26 it shall be allocated. If its **designator** has more than one **part-ref**, the rightmost **part-ref** shall
27 have nonzero **corank**.

28 DIM (optional) shall be an integer scalar with a value in the range $1 \leq \text{DIM} \leq n$, where n is the **corank**
29 of COARRAY. The corresponding **actual argument** shall not be an optional dummy argument,
30 a disassociated pointer, or an unallocated allocatable.

31 KIND (optional) shall be a scalar integer **constant expression**.

32 4 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the value
33 of KIND; otherwise, the kind type parameter is that of default integer type. The result is scalar if DIM is
34 present; otherwise, the result is an array of **rank** one and size n , where n is the **corank** of COARRAY.

35 5 **Result Value.**

1 *Case (i):* If DIM is present, the result has a value equal to the lower **cobound** for **codimension** DIM of
2 COARRAY.

3 *Case (ii):* If DIM is absent, the result has a value whose i^{th} element is equal to the lower **cobound** for
4 **codimension** i of COARRAY, for $i = 1, 2, \dots, n$, where n is the **corank** of COARRAY.

5 6 **Examples.** If A is allocated by the statement ALLOCATE (A [2:3, 7:*) then LCOBOUND (A) is [2, 7] and
6 LCOBOUND (A, DIM=2) is 7.

7 **16.9.121 LEADZ (I)**

8 1 **Description.** Number of leading zero bits.

9 2 **Class.** **Elemental** function.

10 3 **Argument.** I shall be of type integer.

11 4 **Result Characteristics.** Default integer.

12 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
13 value $\text{BIT\_SIZE (I)} - 1 - k$, where k is the position of the leftmost 1 bit in I. The model for the interpretation
14 of an integer value as a sequence of bits is in 16.3.

15 6 **Examples.** LEADZ (1) has the value 31 if BIT_SIZE (1) has the value 32.

16 **16.9.122 LEN (STRING [, KIND])**

17 1 **Description.** Length of a character entity.

18 2 **Class.** **Inquiry function**.

19 3 **Arguments.**

20 STRING shall be of type character. If it is an unallocated **allocatable** variable or a pointer that is not
21 associated, its **length type parameter** shall not be **deferred**.

22 KIND (optional) shall be a scalar integer **constant expression**.

23 4 **Result Characteristics.** Integer scalar. If KIND is present, the kind type parameter is that specified by the
24 value of KIND; otherwise the kind type parameter is that of default integer type.

25 5 **Result Value.** The result has a value equal to the number of characters in STRING if it is scalar or in an
26 element of STRING if it is an array.

27 6 **Example.** If C is declared by the statement

28 7 CHARACTER (11) C (100)

29 8 LEN (C) has the value 11.

30 **16.9.123 LEN_TRIM (STRING [, KIND])**

31 1 **Description.** Length without trailing blanks.

32 2 **Class.** **Elemental** function.

33 3 **Arguments.**

1 STRING shall be of type character.
 2 KIND (optional) shall be a scalar integer [constant expression](#).

3 4 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the value
 4 of KIND; otherwise the kind type parameter is that of default integer type.

5 5 **Result Value.** The result has a value equal to the number of characters remaining after any trailing blanks
 6 in STRING are removed. If the argument contains no nonblank characters, the result is zero.

7 6 **Examples.** LEN_TRIM (' A B ') has the value 4 and LEN_TRIM (' ') has the value 0.

8 **16.9.124 LGE (STRING_A, STRING_B)**

9 1 **Description.** ASCII greater than or equal.

10 2 **Class.** [Elemental](#) function.

11 3 **Arguments.**

12 STRING_A shall be default character or [ASCII character](#).

13 STRING_B shall be of type character with the same kind type parameter as STRING_A.

14 4 **Result Characteristics.** Default logical.

15 5 **Result Value.** If the strings are of unequal length, the comparison is made as if the shorter string were
 16 extended on the right with blanks to the length of the longer string. If either string contains a character not
 17 in the [ASCII character](#) set, the result is processor dependent. The result is true if the strings are equal or if
 18 STRING_A follows STRING_B in the [ASCII collating sequence](#); otherwise, the result is false.

NOTE

The result is true if both STRING_A and STRING_B are of zero length.

19 6 **Example.** LGE ('ONE', 'TWO') has the value false.

20 **16.9.125 LGT (STRING_A, STRING_B)**

21 1 **Description.** ASCII greater than.

22 2 **Class.** [Elemental](#) function.

23 3 **Arguments.**

24 STRING_A shall be default character or [ASCII character](#).

25 STRING_B shall be of type character with the same kind type parameter as STRING_A.

26 4 **Result Characteristics.** Default logical.

27 5 **Result Value.** If the strings are of unequal length, the comparison is made as if the shorter string were
 28 extended on the right with blanks to the length of the longer string. If either string contains a character
 29 not in the [ASCII character](#) set, the result is processor dependent. The result is true if STRING_A follows
 30 STRING_B in the [ASCII collating sequence](#); otherwise, the result is false.

NOTE

The result is false if both STRING_A and STRING_B are of zero length.

1 6 **Example.** LGT ('ONE', 'TWO') has the value false.

2 **16.9.126 LLE (STRING_A, STRING_B)**

3 1 **Description.** ASCII less than or equal.

4 2 **Class.** [Elemental](#) function.

5 3 **Arguments.**

6 STRING_A shall be default character or [ASCII character](#).

7 STRING_B shall be of type character with the same kind type parameter as STRING_A.

8 4 **Result Characteristics.** Default logical.

9 5 **Result Value.** If the strings are of unequal length, the comparison is made as if the shorter string were
10 extended on the right with blanks to the length of the longer string. If either string contains a character not
11 in the [ASCII character](#) set, the result is processor dependent. The result is true if the strings are equal or if
12 STRING_A precedes STRING_B in the [ASCII collating sequence](#); otherwise, the result is false.

NOTE

The result is true if both STRING_A and STRING_B are of zero length.

13 6 **Example.** LLE ('ONE', 'TWO') has the value true.

14 **16.9.127 LLT (STRING_A, STRING_B)**

15 1 **Description.** ASCII less than.

16 2 **Class.** [Elemental](#) function.

17 3 **Arguments.**

18 STRING_A shall be default character or [ASCII character](#).

19 STRING_B shall be of type character with the same kind type parameter as STRING_A.

20 4 **Result Characteristics.** Default logical.

21 5 **Result Value.** If the strings are of unequal length, the comparison is made as if the shorter string were
22 extended on the right with blanks to the length of the longer string. If either string contains a character
23 not in the [ASCII character](#) set, the result is processor dependent. The result is true if STRING_A precedes
24 STRING_B in the [ASCII collating sequence](#); otherwise, the result is false.

NOTE

The result is false if both STRING_A and STRING_B are of zero length.

25 6 **Example.** LLT ('ONE', 'TWO') has the value true.

16.9.128 LOG (X)

Description. Natural logarithm.

Class. [Elemental](#) function.

Argument. X shall be of type real or complex. If X is real, its value shall be greater than zero. If X is complex, its value shall not be zero.

Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to $\log_e X$. A result of type complex is the principal value with imaginary part ω in the range $-\pi \leq \omega \leq \pi$. 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 does not distinguish between positive and negative real zero, and approximately $-\pi$ if the imaginary part of X is negative real zero.

Example. LOG (10.0) has the value 2.3025851 (approximately).

16.9.129 LOG_GAMMA (X)

Description. Logarithm of the absolute value of the gamma function.

Class. [Elemental](#) function.

Argument. X shall be of type real. Its value shall not be a negative integer or zero.

Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to the natural logarithm of the absolute value of the gamma function of X.

Example. LOG_GAMMA (3.0) has the value 0.693 (approximately).

16.9.130 LOG10 (X)

Description. Common logarithm.

Class. [Elemental](#) function.

Argument. X shall be of type real. The value of X shall be greater than zero.

Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to $\log_{10} X$.

Example. LOG10 (10.0) has the value 1.0 (approximately).

16.9.131 LOGICAL (L [, KIND])

Description. Conversion between kinds of logical.

Class. [Elemental](#) function.

Arguments.

L shall be of type logical.

1 KIND (optional) shall be a scalar integer [constant expression](#).

2 4 **Result Characteristics.** Logical. If KIND is present, the kind type parameter is that specified by the value
3 of KIND; otherwise, the kind type parameter is that of default logical.

4 5 **Result Value.** The value is that of L.

5 6 **Example.** LOGICAL (L [.OR. .NOT](#) L) has the value true and is default logical, regardless of the kind type
6 parameter of the logical variable L.

7 16.9.132 MASKL (I [, KIND])

8 1 **Description.** Left justified mask.

9 2 **Class.** [Elemental](#) function.

10 3 **Arguments.**

11 I shall be of type integer. It shall be nonnegative and less than or equal to the number of bits z
12 of the model integer defined for bit manipulation contexts in [16.3](#) for the kind of the result.

13 KIND (optional) shall be a scalar integer [constant expression](#).

14 4 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the value
15 of KIND; otherwise, the kind type parameter is that of default integer type.

16 5 **Result Value.** The result value has its leftmost I bits set to 1 and the remaining bits set to 0. The model for
17 the interpretation of an integer value as a sequence of bits is in [16.3](#).

18 6 **Example.** MASKL (3) has the value SHIFTL (7, BIT_SIZE (0) – 3).

19 16.9.133 MASKR (I [, KIND])

20 1 **Description.** Right justified mask.

21 2 **Class.** [Elemental](#) function.

22 3 **Arguments.**

23 I shall be of type integer. It shall be nonnegative and less than or equal to the number of bits z
24 of the model integer defined for bit manipulation contexts in [16.3](#) for the kind of the result.

25 KIND (optional) shall be a scalar integer [constant expression](#).

26 4 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the value
27 of KIND; otherwise, the kind type parameter is that of default integer type.

28 5 **Result Value.** The result value has its rightmost I bits set to 1 and the remaining bits set to 0. The model
29 for the interpretation of an integer value as a sequence of bits is in [16.3](#).

30 6 **Example.** MASKR (3) has the value 7.

31 16.9.134 MATMUL (MATRIX_A, MATRIX_B)

32 1 **Description.** Matrix multiplication.

33 2 **Class.** [Transformational](#) function.

1 3 **Arguments.**

2 MATRIX_A shall be a rank-one or rank-two array of **numeric type** or logical type.

3 MATRIX_B shall be of **numeric type** if MATRIX_A is of **numeric type** and of logical type if MATRIX_A is of
4 logical type. It shall be an array of **rank** one or two. MATRIX_A and MATRIX_B shall not both
5 have **rank** one. The size of the first (or only) dimension of MATRIX_B shall equal the size of
6 the last (or only) dimension of MATRIX_A.

7 4 **Result Characteristics.** If the arguments are of **numeric type**, the type and kind type parameter of the res-
8 ult are determined by the types of the arguments as specified in 10.1.9.3 for the * operator. If the arguments
9 are of type logical, the result is of type logical with the kind type parameter of the arguments as specified in
10 10.1.9.3 for the .AND. operator. The shape of the result depends on the shapes of the arguments as follows:

11 *Case (i):* If MATRIX_A has shape $[n, m]$ and MATRIX_B has shape $[m, k]$, the result has shape $[n, k]$.

12 *Case (ii):* If MATRIX_A has shape $[m]$ and MATRIX_B has shape $[m, k]$, the result has shape $[k]$.

13 *Case (iii):* If MATRIX_A has shape $[n, m]$ and MATRIX_B has shape $[m]$, the result has shape $[n]$.

14 5 **Result Value.**

15 *Case (i):* Element (i, j) of the result has the value SUM (MATRIX_A $(i, :)$ * MATRIX_B $(:, j)$) if the argu-
16 ments are of **numeric type** and has the value ANY (MATRIX_A $(i, :)$.AND. MATRIX_B $(:, j)$) if
17 the arguments are of logical type.

18 *Case (ii):* Element (j) of the result has the value SUM (MATRIX_A $(:)$ * MATRIX_B $(:, j)$) if the arguments
19 are of **numeric type** and has the value ANY (MATRIX_A $(:)$.AND. MATRIX_B $(:, j)$) if the argu-
20 ments are of logical type.

21 *Case (iii):* Element (i) of the result has the value SUM (MATRIX_A $(i, :)$ * MATRIX_B $(:)$) if the arguments
22 are of **numeric type** and has the value ANY (MATRIX_A $(i, :)$.AND. MATRIX_B $(:)$) if the argu-
23 ments are of logical type.

24 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
25 [1, 2, 3].

26 *Case (i):* The result of MATMUL (A, B) is the matrix-matrix product AB with the value $\begin{bmatrix} 14 & 20 \\ 20 & 29 \end{bmatrix}$.

27 *Case (ii):* The result of MATMUL (X, A) is the vector-matrix product XA with the value [5, 8, 11].

28 *Case (iii):* The result of MATMUL (A, Y) is the matrix-vector product AY with the value [14, 20].

29 **16.9.135 MAX (A1, A2 [, A3, ...])**

30 1 **Description.** Maximum value.

31 2 **Class.** **Elemental** function.

32 3 **Arguments.** The arguments shall all have the same type which shall be integer, real, or character and they
33 shall all have the same kind type parameter.

34 4 **Result Characteristics.** The type and kind type parameter of the result are the same as those of the argu-
35 ments. For arguments of character type, the length of the result is the length of the longest argument.

36 5 **Result Value.** The value of the result is that of the largest argument. For arguments of character type,
37 the result is the value that would be selected by application of intrinsic relational operators; that is, the

[collating sequence](#) for characters with the kind type parameter of the arguments is applied. If the selected argument is shorter than the longest argument, the result is extended with blanks on the right to the length of the longest argument.

Examples. MAX (−9.0, 7.0, 2.0) has the value 7.0, MAX ('Z', 'BB') has the value 'Z', and MAX (['A', 'Z'], ['BB', 'Y ']) has the value ['BB', 'Z '].

16.9.136 MAXEXPONENT (X)

Description. Maximum exponent of a real model.

Class. [Inquiry function](#).

Argument. X shall be of type real. It may be a scalar or an array.

Result Characteristics. Default integer scalar.

Result Value. The result has the value $e_{\max}$, as defined in [16.4](#) for the model representing numbers of the same type and kind type parameter as X.

Example. MAXEXPONENT (X) has the value 127 for real X whose model is as in [16.4](#), [NOTE](#).

16.9.137 MAXLOC (ARRAY, DIM [, MASK, KIND, BACK]) or MAXLOC (ARRAY [, MASK, KIND, BACK])

Description. Location(s) of maximum value.

Class. [Transformational function](#).

Arguments.

ARRAY shall be an array of type integer, real, or character.

DIM shall be an integer scalar with a value in the range $1 \leq \text{DIM} \leq n$, 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.

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 $[d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n]$, where $[d_1, d_2, \dots, d_n]$ is the shape of ARRAY.

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 $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$ of the result is equal to MAXLOC (ARRAY $(s_1, s_2, \dots, s_{\text{DIM}-1}, \cdot, s_{\text{DIM}+1}, \dots, s_n)$, DIM = 1 [, MASK = MASK $(s_1, s_2, \dots, s_{\text{DIM}-1}, \cdot, s_{\text{DIM}+1}, \dots, s_n)$, KIND = KIND, BACK = BACK]).

If only one element has the maximum value, that element's subscripts are returned. Otherwise, if more than one element has the maximum value and BACK is absent or present with the value false, the element whose subscripts are returned is the first such element, taken in array element order. If BACK is present with the value true, the element whose subscripts are returned is the last such element, taken in array element order.

If ARRAY has type character, the result is the value that would be selected by application of intrinsic relational operators; that is, the **collating sequence** for characters with the kind type parameter of the arguments is applied.

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.

16.9.138 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 $1 \leq \text{DIM} \leq n$, 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 $[d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n]$ where $[d_1, d_2, \dots, d_n]$ 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 $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$ of the result is equal to

$$\text{MAXVAL}(\text{ARRAY}(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n) \\ [, \text{MASK} = \text{MASK}(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)]).$$

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].

16.9.139 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 type and **type parameters** as TSOURCE. Because TSOURCE and FSOURCE are required to have the same type and **type parameters** (for both the **declared** and **dynamic** types), the result is **polymorphic** if and only if both TSOURCE and FSOURCE are **polymorphic**.

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 $\begin{bmatrix} \text{T} & \cdot & \text{T} \\ \cdot & \cdot & \text{T} \end{bmatrix}$, 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.

16.9.140 MERGE_BITS (I, J, MASK)

Description. Merge of bits under mask.

Class. [Elemental](#) function.

Arguments.

I shall be of type integer or a [boz-literal-constant](#).

J shall be of type integer or a [boz-literal-constant](#). If both I and J are of type integer they shall have the same kind type parameter. I and J shall not both be [boz-literal-constants](#).

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.

Result Characteristics. Same as I if I is of type integer; otherwise, same as J.

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))).

Example. MERGE_BITS (13, 18, 22) has the value 4.

16.9.141 MIN (A1, A2 [, A3, ...])

Description. Minimum value.

Class. [Elemental](#) function.

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.

Result Characteristics. The type and kind type parameter of the result are the same as those of the arguments. For arguments of character type, the length of the result is the length of the longest argument.

Result Value. The value of the result is that of the 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.

Examples. MIN (−9.0, 7.0, 2.0) has the value −9.0, MIN ('A', 'YY') has the value 'A', and MIN (['Z', 'A'], ['YY', 'B ']) has the value ['YY', 'A '].

16.9.142 MINEXPONENT (X)

Description. Minimum exponent of a real model.

Class. [Inquiry function](#).

Argument. X shall be of type real. It may be a scalar or an array.

Result Characteristics. Default integer scalar.

Result Value. The result has the value $e_{\min}$, as defined in [16.4](#) for the model representing numbers of the same type and kind type parameter as X.

1 6 **Example.** MINEXPONENT (X) has the value -126 for real X whose model is as in 16.4, NOTE.

2 16.9.143 MINLOC (ARRAY, DIM [, MASK, KIND, BACK]) or MINLOC (ARRAY [, MASK, KIND, BACK])

3 1 **Description.** Location(s) of minimum value.

4 2 **Class.** Transformational function.

5 3 **Arguments.**

6 ARRAY shall be an array of type integer, real, or character.

7 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.

8 MASK (optional) shall be of type logical and shall be conformable with ARRAY.

9 KIND (optional) shall be a scalar integer constant expression.

10 BACK (optional) shall be a logical scalar.

11 4 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the value of
12 KIND; otherwise the kind type parameter is that of default integer type. If DIM does not appear, the result
13 is an array of rank one and of size equal to the rank of ARRAY; otherwise, the result is of rank $n - 1$ and
14 shape $[d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n]$, where $[d_1, d_2, \dots, d_n]$ is the shape of ARRAY.

15 5 **Result Value.**

16 *Case (i):* If DIM does not appear and MASK is absent the result is a rank-one array whose element values
17 are the values of the subscripts of an element of ARRAY whose value equals the minimum value
18 of all the elements of ARRAY. The i^{th} subscript returned lies in the range 1 to e_i , where e_i is
19 the extent of the i^{th} dimension of ARRAY. If ARRAY has size zero, all elements of the result are
20 zero.

21 *Case (ii):* If DIM does not appear and MASK is present, the result is a rank-one array whose element val-
22 ues are the values of the subscripts of an element of ARRAY, corresponding to a true element
23 of MASK, whose value equals the minimum value of all such elements of ARRAY. The i^{th} sub-
24 script returned lies in the range 1 to e_i , where e_i is the extent of the i^{th} dimension of ARRAY.
25 If ARRAY has size zero or every element of MASK has the value false, all elements of the result
26 are zero.

27 *Case (iii):* If ARRAY has rank one and DIM is specified, the result has a value equal to that of the first
28 element of MINLOC (ARRAY [, MASK = MASK, KIND = KIND, BACK = BACK]). Otherwise, if
29 DIM is specified, the value of element $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$ of the result is equal to

MINLOC (ARRAY ($s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n$),

DIM = 1

30 $[, \text{MASK} = \text{MASK} (s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n),$

KIND = KIND,

BACK = BACK]).

31 6 If only one element has the minimum value, that element's subscripts are returned. Otherwise, if more than
32 one element has the minimum value and BACK is absent or present with the value false, the element whose
33 subscripts are returned is the first such element, taken in array element order. If BACK is present with
34 the value true, the element whose subscripts are returned is the last such element, taken in array element
35 order.

36 7 If ARRAY is of type character, the result is the value that would be selected by application of intrinsic re-
37 lational operators; that is, the collating sequence for characters with the kind type parameter of the argu-

ments 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.

16.9.144 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 $[d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n]$ where $[d_1, d_2, \dots, d_n]$ 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 $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$ of the result is equal to

$$\text{MINVAL} (\text{ARRAY} (s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n) \\ [, \text{MASK} = \text{MASK} (s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)]).$$

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.

1 *Case (i):* The value of MINVAL ([1, 2, 3]) is 1.

2 *Case (ii):* MINVAL (C, MASK = C > 0.0) is the minimum of the positive elements of C.

3 *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].

4 **16.9.145 MOD (A, P)**

5 1 **Description.** Remainder function.

6 2 **Class.** [Elemental](#) function.

7 3 **Arguments.**

8 A shall be of type integer or real.

9 P shall be of the same type and kind type parameter as A. P shall not be zero.

10 4 **Result Characteristics.** Same as A.

11 5 **Result Value.** The value of the result is $A - \text{INT} (A/P) * P$.

12 6 **Examples.** MOD (3.0, 2.0) has the value 1.0 (approximately). MOD (8, 5) has the value 3. MOD (−8, 5) has
13 the value −3. MOD (8, −5) has the value 3. MOD (−8, −5) has the value −3.

14 **16.9.146 MODULO (A, P)**

15 1 **Description.** Modulo function.

16 2 **Class.** [Elemental](#) function.

17 3 **Arguments.**

18 A shall be of type integer or real.

19 P shall be of the same type and kind type parameter as A. P shall not be zero.

20 4 **Result Characteristics.** Same as A.

21 5 **Result Value.**

22 *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
23 integer, the inequalities $0 \leq R < P$ hold if $P > 0$, and $P < R \leq 0$ hold if $P < 0$.

24 *Case (ii):* A is of type real. The value of the result is $A - \text{FLOOR} (A / P) * P$.

25 6 **Examples.** MODULO (8, 5) has the value 3. MODULO (−8, 5) has the value 2. MODULO (8, −5) has the
26 value −2. MODULO (−8, −5) has the value −3.

27 **16.9.147 MOVE_ALLOC (FROM, TO [, STAT, ERRMSG])**

28 1 **Description.** Move an allocation.

29 2 **Class.** Subroutine, [simple](#) if and only if FROM is not a [coarray](#).

30 3 **Arguments.**

31 FROM may be of any type, [rank](#), and [corank](#). It shall be [allocatable](#) and shall not be a [coindexed object](#).
32 If it is a [coarray](#), the FROM arguments in corresponding invocations of MOVE_ALLOC shall be
33 corresponding [coarrays](#). It is an [INTENT \(INOUT\)](#) argument.

- 1 TO shall be [type compatible](#) (7.3.3) with FROM and have the same [rank](#) and [corank](#). It shall be
 2 [allocatable](#) and shall not be a [coindexed object](#). It shall be [polymorphic](#) if FROM is [polymorphic](#).
 3 If it is a [coarray](#), the TO arguments in corresponding invocations of MOVE_ALLOC shall be
 4 corresponding [coarrays](#). It is an [INTENT \(OUT\)](#) argument. Each nondeferred parameter of the
 5 [declared type](#) of TO shall have the same value as the corresponding parameter of the [declared](#)
 6 [type](#) of FROM.
- 7 STAT (optional) shall be a noncoindexed integer scalar with a decimal exponent range of at least four. It is
 8 an [INTENT \(OUT\)](#) argument.
- 9 ERRMSG (optional) shall be a noncoindexed default character scalar. It is an [INTENT \(INOUT\)](#) argument.
- 10 4 If execution of MOVE_ALLOC is successful, or if [STAT_FAILED_IMAGE](#) is assigned to STAT,
- 11 • On invocation of MOVE_ALLOC, if the allocation status of TO is allocated, it is deallocated. Then, if
 12 FROM has an allocation status of allocated on entry to MOVE_ALLOC, TO becomes allocated with
 13 [dynamic type](#), [type parameters](#), [bounds](#), [cobounds](#), and value identical to those that FROM had on
 14 entry to MOVE_ALLOC. Note that if FROM and TO are the same variable, it shall be unallocated when
 15 MOVE_ALLOC is invoked.
 - 16 • If TO has the [TARGET attribute](#), any pointer [associated](#) with FROM on entry to MOVE_ALLOC be-
 17 comes correspondingly associated with TO. If TO does not have the [TARGET attribute](#), the [pointer](#)
 18 [association](#) status of any pointer [associated](#) with FROM on entry becomes undefined.
 - 19 • The allocation status of FROM becomes unallocated.
- 20 5 When a reference to MOVE_ALLOC is executed for which the FROM argument is a [coarray](#), there is an im-
 21 plicit synchronization of all [active images](#) of the [current team](#). On those [images](#), execution of the [segment](#)
 22 (11.7.2) following the [CALL statement](#) is delayed until all other [active images](#) of the [current team](#) have ex-
 23 ecuted the same statement the same number of times. When such a reference is executed, if any [image](#) of
 24 the [current team](#) has [stopped](#) or [failed](#), an error condition occurs.
- 25 6 If STAT is present and execution is successful, it is assigned the value zero.
- 26 7 If an error condition occurs,
- 27 • if STAT is absent, [error termination](#) is initiated;
 - 28 • otherwise, if FROM is a [coarray](#) and the [current team](#) contains a [stopped image](#), STAT is assigned the
 29 value [STAT_STOPPED_IMAGE](#) from the intrinsic module [ISO_FORTRAN_ENV](#);
 - 30 • otherwise, if FROM is a [coarray](#) and the [current team](#) contains a [failed image](#), and no other error
 31 condition occurs, STAT is assigned the value [STAT_FAILED_IMAGE](#) from the intrinsic module [ISO_-](#)
 32 [FORTRAN_ENV](#);
 - 33 • otherwise, STAT is assigned a processor-dependent positive value that differs from that of [STAT_-](#)
 34 [STOPPED_IMAGE](#) or [STAT_FAILED_IMAGE](#).
- 35 8 If the ERRMSG argument is present and an error condition occurs, it is assigned an explanatory message.
 36 If no error condition occurs, the definition status and value of ERRMSG are unchanged.
- 37 9 **Example.** The example below demonstrates reallocation of GRID to twice its previous size, with its previ-
 38 ous contents evenly distributed over the new elements so that intermediate points can be inserted.

```

39 REAL, ALLOCATABLE :: GRID(:), TEMPGRID(:)
40 ...
41 ALLOCATE (GRID(-N:N))           ! initial allocation of GRID
42 ...

```

```

1      ALLOCATE(TEMPGRID(-2*N:2*N)) ! allocate bigger grid
2      TEMPGRID(:,2)=GRID ! distribute values to new locations
3      CALL MOVE_ALLOC(TO=GRID, FROM=TEMPGRID)

```

4 The old grid is deallocated because TO is **INTENT (OUT)**, and GRID then takes over the new grid allocation.

NOTE

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.

5 16.9.148 MVBITS (FROM, FROMPOS, LEN, TO, TOPOS)

6 1 **Description.** Copy a sequence of bits.

7 2 **Class.** **Simple elemental** subroutine.

8 3 **Arguments.**

9	FROM	shall be of type integer. It is an INTENT (IN) argument.
10	FROMPOS	shall be of type integer and nonnegative. It is an INTENT (IN) argument. FROMPOS + LEN
11		shall be less than or equal to BIT_SIZE (FROM). The model for the interpretation of an integer
12		value as a sequence of bits is in 16.3 .
13	LEN	shall be of type integer and nonnegative. It is an INTENT (IN) argument.
14	TO	shall be a variable of the same type and kind type parameter value as FROM and may be asso-
15		ciated with FROM (15.9.3). It is an INTENT (INOUT) argument. TO is defined by copying the
16		sequence of bits of length LEN, starting at position FROMPOS of FROM to position TOPOS of
17		TO. No other bits of TO are altered. On return, the LEN bits of TO starting at TOPOS are equal
18		to the value that the LEN bits of FROM starting at FROMPOS had on entry. The model for the
19		interpretation of an integer value as a sequence of bits is in 16.3 .
20	TOPOS	shall be of type integer and nonnegative. It is an INTENT (IN) argument. TOPOS + LEN shall
21		be less than or equal to BIT_SIZE (TO).

22 4 **Example.** If TO has the initial value 6, its value after the statement CALL MVBITS (7, 2, 2, TO, 0) is 5.

23 16.9.149 NEAREST (X, S)

24 1 **Description.** Adjacent machine number.

25 2 **Class.** **Elemental** function.

26 3 **Arguments.**

27	X	shall be of type real.
28	S	shall be of type real and not equal to zero.

29 4 **Result Characteristics.** Same as X.

30 5 **Result Value.** The result has a value equal to the machine-representable number distinct from X and

31 nearest to it in the direction of the ∞ with the same sign as S.

32 6 **Example.** NEAREST (3.0, 2.0) has the value $3 + 2^{-22}$ on a machine whose representation for default real is

33 that of the model in **16.4**, **NOTE**.

NOTE

Unlike other floating-point manipulation functions, NEAREST operates on machine-representable numbers rather than model numbers. On many systems there are machine-representable numbers that lie between adjacent model numbers.

16.9.150 NEW_LINE (A)

Description. Newline character.

Class. [Inquiry function](#).

Argument. A shall be of type character. It may be a scalar or an array.

Result Characteristics. Character scalar of length one with the same kind type parameter as A.

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.

Example. If there is a suitable newline character, and unit 10 is connected for formatted stream output, the statement

```
WRITE (10, '(A) ') 'New'//NEW_LINE('a')//'Line'
```

will write a record containing “New” and then a record containing “Line”.

16.9.151 NEXT (A [, STAT])

Description. Next enumeration value.

Class. [Elemental function](#).

Arguments.

A shall be of enumeration type.

STAT (optional) shall be an integer scalar with a decimal exponent range of at least four. It is an [INTENT \(OUT\)](#) argument. If A is equal to the last enumerator of its type, it is assigned a processor-dependent positive value; otherwise, it is assigned the value zero. If STAT would have been assigned a nonzero value but is not present, [error termination](#) is initiated.

Result Characteristics. Same as A.

Result Value. If A is equal to the last enumerator of its type, the value of the result is that of A. Otherwise, the value of the result is the next enumerator following the value of A.

Example. If the enumerators of an enumeration type are EN1, EN2, EN3, and EN4, NEXT (EN1) is equal to EN2, and NEXT (EN4, ISTAT) is equal to EN4 and a positive value is assigned to ISTAT.

16.9.152 NINT (A [, KIND])

1 **Description.** Nearest integer.

2 **Class.** Elemental function.

3 **Arguments.**

A shall be of type real.

KIND (optional) shall be a scalar integer constant expression.

4 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default integer type.

5 **Result Value.** The result is the integer nearest A, or if there are two integers equally near A, the result is whichever such integer has the greater magnitude.

6 **Example.** NINT (2.783) has the value 3.

16.9.153 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 $[d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n]$, where n is the rank of X and $[d_1, d_2, \dots, d_n]$ 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 $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$ of the result is equal to NORM2 $(X(s_1, s_2, \dots, s_{\text{DIM}-1}, \vdots, s_{\text{DIM}+1}, \dots, s_n))$.

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).

16.9.154 NOT (I)

1 **Description.** Bitwise complement.

2 **Class.** Elemental function.

- 1 3 **Argument.** I shall be of type integer.
- 2 4 **Result Characteristics.** Same as I.
- 3 5 **Result Value.** The result has the value obtained by complementing I bit-by-bit according to Table 16.8.

Table 16.8 — Bitwise result table for NOT

I	NOT (I)
1	0
0	1

- 4 6 The model for the interpretation of an integer value as a sequence of bits is in 16.3.
- 5 7 **Example.** If I is represented by the string of bits 01010101, NOT (I) has the binary value 10101010.

6 16.9.155 NULL ([MOLD])

- 7 1 **Description.** [Disassociated](#) pointer or unallocated [allocatable](#) entity.
- 8 2 **Class.** [Transformational function](#).
- 9 3 **Argument.** MOLD shall be a pointer or [allocatable](#). It may be of any type or may be a [procedure pointer](#). If
 10 MOLD is a pointer its [pointer association](#) status may be undefined, [disassociated](#), or associated. If MOLD is
 11 [allocatable](#) its allocation status may be allocated or unallocated. It need not be defined with a value.
- 12 4 **Result Characteristics.** If MOLD is present, the [characteristics](#) are the same as MOLD. If MOLD has [deferred](#)
 13 [type parameters](#), those type parameters of the result are [deferred](#).
- 14 5 If MOLD is absent, the [characteristics](#) of the result are determined by the entity with which the reference
 15 is associated. See Table 16.9. MOLD shall not be absent in any other context. If any type parameters of the
 16 contextual entity are [deferred](#), those type parameters of the result are [deferred](#). If any type parameters of
 17 the contextual entity are assumed, MOLD shall be present.
- 18 6 If the context of the reference to NULL is an [actual argument](#) in a generic procedure reference, MOLD shall
 19 be present if the type, type parameters, or [rank](#) are required to resolve the generic reference. If the context
 20 of the reference to NULL is an [actual argument](#) corresponding to an [assumed-rank dummy argument](#), MOLD
 21 shall be present.

Table 16.9 — 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 entity in a declaration	the entity
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

- 22 7 **Result.** The result is a [disassociated](#) pointer or an unallocated [allocatable](#) entity.
- 23 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

```

16.9.156 NUM_IMAGES () or NUM_IMAGES (TEAM) or NUM_IMAGES (TEAM_NUMBER)

1 **Description.** Number of [images](#).

2 **Class.** [Transformational function](#).

3 **Arguments.**

TEAM shall be a scalar of type [TEAM_TYPE](#) from the intrinsic module [ISO_FORTRAN_ENV](#), with a value that identifies the [current](#) or an [ancestor team](#).

TEAM_NUMBER shall be an integer scalar. It shall identify the [initial team](#) or a [sibling team](#) of the [current team](#).

4 **Result Characteristics.** Default integer scalar.

5 **Result Value.** The number of [images](#) in the specified [team](#), or in the [current team](#) if no team is specified.

6 **Example.** The following code uses [image](#) 1 to read data and broadcast it to other [images](#).

```

REAL :: P[*]
IF (THIS_IMAGE()==1) THEN
  READ (6,*) P
  DO I = 2, NUM_IMAGES()
    P[I] = P
  END DO
END IF
SYNC ALL

```

16.9.157 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.

1 MOLD shall be an integer or real scalar. If it is a variable, it need not be defined.
 2 ROUND (optional) shall be a logical scalar. ROUND shall appear only if X is of type real and MOLD is of type
 3 integer.

4 4 **Result Characteristics.** Default logical.

5 5 **Result Value.**

6 *Case (i):* If MOLD is of type integer, and ROUND is absent or present with the value false, the result
 7 is true if and only if the value of X is an IEEE infinity or NaN, or if the integer with largest
 8 magnitude that lies between zero and X inclusive is not representable by objects with the type
 9 and kind of MOLD.

10 *Case (ii):* If MOLD is of type integer, and ROUND is present with the value true, the result is true if and
 11 only if the value of X is an IEEE infinity or NaN, or if the integer nearest X, or the integer of
 12 greater magnitude if two integers are equally near to X, is not representable by objects with
 13 the type and kind of MOLD.

14 *Case (iii):* Otherwise, the result is true if and only if the value of X is an IEEE infinity or NaN that is not
 15 supported by objects of the type and kind of MOLD, or if X is a finite number and the result of
 16 rounding the value of X (according to the IEEE rounding mode if appropriate) to the extended
 17 model for the kind of MOLD has magnitude larger than that of the largest finite number with
 18 the same sign as X that is representable by objects with the type and kind of MOLD.

19 6 **Examples.** If INT8 is the kind value for an 8-bit binary integer type, OUT_OF_RANGE (−128.5, 0_INT8) will
 20 have the value false and OUT_OF_RANGE (−128.5, 0_INT8, .TRUE.) will have the value true.

NOTE

MOLD is required to be a scalar because the only information taken from it is its type and kind. Allowing an array MOLD would require that it be conformable with X. ROUND is scalar because allowing an array rounding mode would have severe performance difficulties on many processors.

21 16.9.158 PACK (ARRAY, MASK [, VECTOR])

22 1 **Description.** Array packed into a vector.

23 2 **Class.** Transformational function.

24 3 **Arguments.**

25 ARRAY shall be an array of any type.

26 MASK shall be of type logical and shall be conformable with ARRAY.

27 VECTOR (optional) shall be of the same type and type parameters as ARRAY and shall have rank one. VEC-
 28 TOR shall have at least as many elements as there are true elements in MASK. If MASK is scalar
 29 with the value true, VECTOR shall have at least as many elements as there are in ARRAY.

30 4 **Result Characteristics.** The result is an array of rank one with the same type and type parameters as
 31 ARRAY. If VECTOR is present, the result size is that of VECTOR; otherwise, the result size is the number t of
 32 true elements in MASK unless MASK is scalar with the value true, in which case the result size is the size of
 33 ARRAY.

34 5 **Result Value.** Element i of the result is the element of ARRAY that corresponds to the i^{th} true element of
 35 MASK, taking elements in array element order, for $i = 1, 2, \dots, t$. If VECTOR is present and has size $n > t$,
 36 element i of the result has the value VECTOR (i), for $i = t + 1, \dots, n$.

1 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
 2 function `PACK`. The result of `PACK (M, MASK = M /= 0)` is `[9, 7]` and the result of `PACK (M, M /= 0, VEC-`
 3 `TOR = [2, 4, 6, 8, 10, 12])` is `[9, 7, 6, 8, 10, 12]`.

4 16.9.159 PARITY (MASK) or PARITY (MASK, DIM)

5 1 **Description.** Array reduced by [.NEQV](#) operation.

6 2 **Class.** [Transformational function](#).

7 3 **Arguments.**

8 MASK shall be a logical array.

9 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.

10 4 **Result Characteristics.** The result is of type logical with the same kind type parameter as MASK. It is
 11 scalar if DIM does not appear; otherwise, the result has [rank](#) $n - 1$ and shape $[d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1},$
 12 $\dots, d_n]$ where $[d_1, d_2, \dots, d_n]$ is the shape of MASK.

13 5 **Result Value.**

14 *Case (i):* The result of `PARITY (MASK)` has the value true if an odd number of the elements of MASK are
 15 true, and false otherwise.

16 *Case (ii):* If MASK has [rank](#) one, `PARITY (MASK, DIM)` is equal to `PARITY (MASK)`. Otherwise, the value of
 17 element $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$ of `PARITY (MASK, DIM)` is equal to `PARITY (MASK (s_1,`
 18 $s_2, \dots, s_{\text{DIM}-1}, \cdot, s_{\text{DIM}+1}, \dots, s_n)).$

19 6 **Examples.**

20 *Case (i):* The value of `PARITY ([T, T, T, F])` is true if T has the value true and F has the value false.

21 *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 PAR-
 22 `ITY (B, DIM=1)` has the value `[F, F, T]` and `PARITY (B, DIM=2)` has the value `[F, T]`.

23 16.9.160 POPCNT (I)

24 1 **Description.** Number of one bits.

25 2 **Class.** [Elemental function](#).

26 3 **Argument.** I shall be of type integer.

27 4 **Result Characteristics.** Default integer.

28 5 **Result Value.** The result value is equal to the number of one bits in the sequence of bits of I. The model for
 29 the interpretation of an integer value as a sequence of bits is in [16.3](#).

30 6 **Examples.** `POPCNT ([1, 2, 3, 4, 5, 6])` has the value `[1, 1, 2, 1, 2, 2]`.

16.9.161 POPPAR (I)

Description. Parity expressed as 0 or 1.

Class. [Elemental](#) function.

Argument. I shall be of type integer.

Result Characteristics. Default integer.

Result Value. POPPAR (I) has the value 1 if POPCNT (I) is odd, and 0 if POPCNT (I) is even.

Examples. POPPAR ([1, 2, 3, 4, 5, 6]) has the value [1, 1, 0, 1, 0, 0].

16.9.162 PRECISION (X)

Description. Decimal precision of a real model.

Class. [Inquiry function](#).

Argument. X shall be of type real or complex. It may be a scalar or an array.

Result Characteristics. Default integer scalar.

Result Value. The result has the value $\text{INT}((p - 1) * \text{LOG}_{10}(b)) + k$, where b and p are as defined in [16.4](#) for the model representing real numbers with the same value for the kind type parameter as X, and where k is 1 if b is an integral power of 10 and 0 otherwise.

Example. PRECISION (X) has the value $\text{INT}(23 * \text{LOG}_{10}(2.)) = \text{INT}(6.92...) = 6$ for real X whose model is as in [16.4](#), [NOTE](#).

16.9.163 PRESENT (A)

Description. Presence of optional argument.

Class. [Inquiry function](#).

Argument. A shall be the name of an optional dummy argument that is accessible in the subprogram in which the PRESENT function reference appears. There are no other requirements on A.

Result Characteristics. Default logical scalar.

Result Value. The result has the value true if A is present ([15.5.2.13](#)) and otherwise has the value false.

16.9.164 PREVIOUS (A [, STAT])

Description. Previous enumeration value.

Class. [Elemental](#) function.

Arguments.

A shall be of enumeration type.

STAT (optional) shall be an integer scalar with a decimal exponent range of at least four. It is an [INTENT \(OUT\)](#) argument. If A is equal to the first enumerator of its type, it is assigned a processor-dependent positive value; otherwise, it is assigned the value zero. If STAT would have been

assigned a nonzero value but is not present, **error termination** is initiated.

4 Result Characteristics. Same as A.

5 Result Value. If A is equal to the first enumerator of its type, the value of the result is that of A. Otherwise, the value of the result is the enumerator preceding the value of A.

6 Example. If the enumerators of an enumeration type are EN1, EN2, EN3, and EN4, PREVIOUS (EN3) is equal to EN2, and PREVIOUS (EN1, ISTAT) is equal to EN1 and a positive value is assigned to ISTAT.

16.9.165 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 $[d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n]$ where $[d_1, d_2, \dots, d_n]$ 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 $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$ of PRODUCT (ARRAY, DIM = DIM [, MASK = MASK]) is equal to

$$\text{PRODUCT}(\text{ARRAY}(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n) \\ [, \text{MASK} = \text{MASK}(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)]).$$

6 Examples.

Case (i): The value of PRODUCT ([1, 2, 3]) is 6.

Case (ii): PRODUCT (C, MASK = C > 0.0) forms the product of the positive elements of C.

Case (iii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$, PRODUCT (B, DIM = 1) is [2, 12, 30] and PRODUCT (B, DIM = 2) is [15, 48].

16.9.166 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.

1 4 **Result Characteristics.** Default integer scalar.

2 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
3 and b are as defined in 16.4 for the model representing numbers of the same type and kind type parameter
4 as X .

5 6 **Example.** RADIX (X) has the value 2 for real X whose model is as in 16.4, NOTE.

6 16.9.167 RANDOM_INIT (REPEATABLE, IMAGE_DISTINCT)

7 1 **Description.** Initialize pseudorandom number generator.

8 2 **Class.** Subroutine.

9 3 **Arguments.**

10 REPEATABLE shall be a logical scalar. It is an INTENT (IN) argument.

11 IMAGE_DISTINCT shall be a logical scalar. It is an INTENT (IN) argument.

12 4 The effect of calling RANDOM_INIT depends on the values of the REPEATABLE and IMAGE_DISTINCT argu-
13 ments:

14 *Case (i):* CALL RANDOM_INIT (REPEATABLE=true, IMAGE_DISTINCT=true) is equivalent to invoking
15 RANDOM_SEED with a processor-dependent value for PUT that is different on every invoking
16 image. In each execution of the program with the same execution environment, if the invoking
17 image index value in the initial team is the same, the value for PUT shall be the same.

18 *Case (ii):* CALL RANDOM_INIT(REPEATABLE=true, IMAGE_DISTINCT=false) is equivalent to invoking
19 RANDOM_SEED with a processor-dependent value for PUT that is the same on every invoking
20 image. In each execution of the program with the same execution environment, the value for
21 PUT shall be the same.

22 *Case (iii):* CALL RANDOM_INIT(REPEATABLE=false, IMAGE_DISTINCT=true) is equivalent to invoking
23 RANDOM_SEED with a processor-dependent value for PUT that is different on every invok-
24 ing image. Different values for PUT shall be used for subsequent invocations, and for each
25 execution of the program.

26 *Case (iv):* CALL RANDOM_INIT(REPEATABLE=false, IMAGE_DISTINCT=false) is equivalent to invoking
27 RANDOM_SEED with a processor-dependent value for PUT that is the same on every invok-
28 ing image. Different values for PUT shall be used for subsequent invocations, and for each
29 execution of the program.

30 5 In each of these cases, a different processor-dependent value for PUT shall result in a different sequence of
31 pseudorandom numbers.

32 6 **Example.** The following statement initializes the pseudorandom number generator of the invoking image
33 so that the pseudorandom number sequence will differ from that of other images that execute a similar
34 statement, and will be different on subsequent execution of the program.

35 CALL RANDOM_INIT (REPEATABLE=.FALSE., IMAGE_DISTINCT=.TRUE.)

16.9.168 RANDOM_NUMBER (HARVEST)

Description. Generate pseudorandom number(s).

Class. Subroutine.

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 $0 \leq x < 1$.

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
```

16.9.169 RANDOM_SEED ([SIZE, PUT, GET])

Description. Pseudorandom number generator control.

Class. Subroutine.

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.

If no argument is present, the processor assigns a processor-dependent value to the seed.

The pseudorandom number generator used by **RANDOM_NUMBER** maintains a seed on each **image** that is updated during the execution of **RANDOM_NUMBER** and that can be retrieved or changed by **RANDOM_INIT** or **RANDOM_SEED**³. 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.

³These three procedures only affect the value of the seed on the invoking **image**.

1 6 **Examples.**

```

2      CALL RANDOM_SEED                                ! Processor-dependent initialization
3      CALL RANDOM_SEED (SIZE = K)                      ! Puts size of seed in K
4      CALL RANDOM_SEED (PUT = SEED (1 : K))            ! Define seed
5      CALL RANDOM_SEED (GET = OLD (1 : K))            ! Read current seed

```

6 **16.9.170 RANGE (X)**7 1 **Description.** Decimal exponent range of a numeric model (16.4).8 2 **Class.** Inquiry function.9 3 **Argument.** X shall be of type integer, real, or complex. It may be a scalar or an array.10 4 **Result Characteristics.** Default integer scalar.11 5 **Result Value.**12 *Case (i):* If X is of type integer, the result has the value `INT (LOG10 (HUGE (X)))`.13 *Case (ii):* If X is of type real, the result has the value `INT (MIN (LOG10 (HUGE (X)), -LOG10 (TINY (X))))`.14 *Case (iii):* If X is of type complex, the result has the value `RANGE (REAL (X))`.

15 6 **Examples.** RANGE (X) has the value 38 for real X whose model is as in 16.4, NOTE, because in this case
 16 `HUGE (X) = (1 - 2-24) × 2127` and `TINY (X) = 2-127`.

17 **16.9.171 RANK (A)**18 1 **Description.** Rank of a data object.19 2 **Class.** Inquiry function.20 3 **Argument.** A shall be a data object of any type.21 4 **Result Characteristics.** Default integer scalar.22 5 **Result Value.** The value of the result is the rank of A.

23 6 **Example.** If X is an assumed-rank dummy argument and its associated effective argument is an array of
 24 rank 3, RANK(X) has the value 3.

25 **16.9.172 REAL (A [, KIND])**26 1 **Description.** Conversion to real type.27 2 **Class.** Elemental function.28 3 **Arguments.**29 A shall be of type integer, real, or complex, or a *boz-literal-constant*.

30 KIND (optional) shall be a scalar integer constant expression.

31 4 **Result Characteristics.** Real.

32 *Case (i):* If A is of type integer or real and KIND is present, the kind type parameter is that specified
 33 by the value of KIND. If A is of type integer or real and KIND is not present, the kind type
 34 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 16.3.3. The interpretation of the bit sequence is processor dependent.

Examples. REAL (−3) has the value −3.0. REAL (Z) has the same kind type parameter and the same value as the real part of the complex variable Z.

16.9.173 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 exactly two arguments; each argument shall be a scalar, nonallocatable, noncoarray, nonpointer, nonpolymorphic, nonoptional dummy data object with the same *declared type* and *type parameters* as ARRAY. If one argument has the *ASYNCHRONOUS*, *TARGET*, or *VALUE* attribute, the other shall have that attribute. Its result shall be a nonpolymorphic scalar and have the same *declared type* and *type parameters* as ARRAY. 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 *declared type* and *type parameters* as ARRAY.

ORDERED (optional) shall be a logical scalar.

4 Result Characteristics. The result is of the same *declared type* and *type parameters* as ARRAY. It is scalar if DIM does not appear; otherwise, the result has *rank* $n - 1$ and shape $[d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n]$ where $[d_1, d_2, \dots, d_n]$ is the shape of ARRAY.

5 Result Value.

Case (i): The result of REDUCE (ARRAY, OPERATION [, IDENTITY = IDENTITY, ORDERED = ORDERED]) over the sequence of values in ARRAY is the result of an iterative process. The initial order of the sequence is array element order. While the sequence has more than one element, each iteration involves the execution of $r = \text{OPERATION}(x, y)$ for adjacent x and y in the sequence, with x immediately preceding y , and the subsequent replacement of x and y with r ; if ORDERED is

present with the value true, x and y shall be the first two elements of the sequence. The process continues until the sequence has only one element which is the value of the reduction. If the initial sequence is empty, the result has the value IDENTITY if IDENTITY is present, and otherwise, **error termination** is initiated.

Case (ii): The result of REDUCE (ARRAY, OPERATION, MASK = MASK [, IDENTITY = IDENTITY, ORDERED = ORDERED]) is as for Case (i) except that the initial sequence is only those elements of ARRAY for which the corresponding elements of MASK are true.

Case (iii): If ARRAY has **rank** one, REDUCE (ARRAY, OPERATION, DIM = DIM [, MASK = MASK, IDENTITY = IDENTITY, ORDERED = ORDERED]) has a value equal to that of REDUCE (ARRAY, OPERATION [, MASK = MASK, IDENTITY = IDENTITY, ORDERED = ORDERED]). Otherwise, the value of element $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$ of REDUCE (ARRAY, OPERATION, DIM = DIM [, MASK = MASK, IDENTITY = IDENTITY, ORDERED = ORDERED]) is equal to

REDUCE (ARRAY $(s_1, s_2, \dots, s_{\text{DIM}-1}, \cdot, s_{\text{DIM}+1}, \dots, s_n)$,
 OPERATION = OPERATION,
 DIM=1
 [, MASK = MASK $(s_1, s_2, \dots, s_{\text{DIM}-1}, \cdot, s_{\text{DIM}+1}, \dots, s_n)$,
 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

If OPERATION is not computationally associative, REDUCE without ORDERED=.TRUE. with the same argument values might not always produce the same result, as the processor can apply the associative law to the evaluation.

16.9.174 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.

16.9.175 RESHAPE (SOURCE, SHAPE [, PAD, ORDER])

Description. Arbitrary shape array construction.

Class. Transformational function.

Arguments.

SOURCE shall be an array of any type. If PAD is absent or of size zero, the size of SOURCE shall be greater than or equal to PRODUCT (SHAPE). The size of the result is the product of the values of the elements of SHAPE.

SHAPE shall be a rank-one integer array. **SIZE** (x), where x is the **actual argument** corresponding to **SHAPE**, shall be a **constant expression** whose value is positive and less than 16. It shall not have an element whose value is negative.

PAD (optional) shall be an array of the same type and type parameters as SOURCE.

ORDER (optional) shall be of type integer; shall have the same shape as SHAPE, and its value shall be a permutation of (1, 2, ..., n), where n is the size of SHAPE. If absent, it is as if it were present with value (1, 2, ..., n).

Result Characteristics. The result is an array of shape SHAPE (that is, SHAPE (RESHAPE (SOURCE, SHAPE, PAD, ORDER)) is equal to SHAPE) with the same type and type parameters as SOURCE.

Result Value. The elements of the result, taken in permuted subscript order ORDER (1), ..., ORDER (n), are those of SOURCE in normal array element order followed if necessary by those of PAD in array element order, followed if necessary by additional copies of PAD in array element order.

Examples. 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 $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 0 & 0 \end{bmatrix}$.

16.9.176 RRSPACING (X)

Description. Reciprocal of relative spacing of model numbers.

Class. Elemental function.

Argument. X shall be of type real.

Result Characteristics. Same as X.

Result Value. The result has the value $|Y \times b^{-e}| \times b^p = \text{ABS}(\text{FRACTION}(Y)) * \text{RADIX}(X) / \text{EPSILON}(X)$, where b , e , and p are as defined in 16.4 for Y, the value nearest to X in the model for real values whose kind type parameter is that of X; if there are two such values, the value of greater absolute value is taken. If X is an IEEE infinity, the result is an IEEE NaN. If X is an IEEE NaN, the result is that NaN.

Example. RRSPACING (−3.0) has the value 0.75×2^{24} for reals whose model is as in 16.4, NOTE.

16.9.177 SAME_TYPE_AS (A, B)

Description. Dynamic type equality test.

Class. Inquiry function.

1 3 **Arguments.**

2 A shall be an object of **extensible declared** type or **unlimited polymorphic**. If it is a **polymorphic**
 3 pointer, it shall not have an undefined association status.

4 B shall be an object of **extensible declared** type or **unlimited polymorphic**. If it is a **polymorphic**
 5 pointer, it shall not have an undefined association status.

6 4 **Result Characteristics.** Default logical scalar.

7 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
 8 A is the same as the **dynamic type** of B. If neither A nor B has **extensible dynamic** type, the result is processor
 9 dependent.

NOTE 1

The **dynamic type** of a **disassociated** pointer or unallocated **allocatable** variable is its **declared type**. An **unlimited polymorphic** entity has no **declared type**.

NOTE 2

The test performed by SAME_TYPE_AS is not the same as the test performed by the type guard **TYPE IS**. The test performed by SAME_TYPE_AS does not consider **kind type parameters**.

10 6 **Example.** Given the declarations and assignments

```
11         TYPE T1
12             REAL C
13         END TYPE
14         TYPE, EXTENDS(T1) :: T2
15         END TYPE
16         CLASS(T1), POINTER :: P, Q, R
17         ALLOCATE (P, Q)
18         ALLOCATE (T2 :: R)
```

19 the value of SAME_TYPE_AS (P, Q) will be true, and the value of SAME_TYPE_AS (P, R) will be false.

20 **16.9.178 SCALE (X, I)**

21 1 **Description.** Real number scaled by radix power.

22 2 **Class.** **Elemental** function.

23 3 **Arguments.**

24 X shall be of type real.

25 I shall be of type integer.

26 4 **Result Characteristics.** Same as X.

27 5 **Result Value.** The result has the value $X \times b^I$, where b is defined in 16.4 for model numbers representing
 28 values of X, provided this result is representable; if not, the result is processor dependent.

29 6 **Example.** SCALE (3.0, 2) has the value 12.0 for reals whose model is as in 16.4, NOTE.

16.9.179 SCAN (STRING, SET [, BACK, KIND])

Description. Character set membership search.

Class. [Elemental](#) function.

Arguments.

STRING shall be of type character.

SET shall be of type character with the same kind type parameter as STRING.

BACK (optional) shall be of type logical.

KIND (optional) shall be a scalar integer [constant expression](#).

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type.

Result Value.

Case (i): If BACK is absent or is present with the value false and if STRING contains at least one character that is in SET, the value of the result is the position of the leftmost character of STRING that is in SET.

Case (ii): If BACK is present with the value true and if STRING contains at least one character that is in SET, the value of the result is the position of the rightmost character of STRING that is in SET.

Case (iii): The value of the result is zero if no character of STRING is in SET or if the length of STRING or SET is zero.

Examples.

Case (i): SCAN ('FORTRAN', 'TR') has the value 3.

Case (ii): SCAN ('FORTRAN', 'TR', BACK = [.TRUE.](#)) has the value 5.

Case (iii): SCAN ('FORTRAN', 'BCD') has the value 0.

16.9.180 SELECTED_CHAR_KIND (NAME)

Description. Character kind selection.

Class. [Transformational](#) function.

Argument. NAME shall be default character scalar.

Result Characteristics. Default integer scalar.

Result Value. If NAME has the value DEFAULT, then the result has a value equal to that of the kind type parameter of default character. If NAME has the value ASCII, then the result has a value equal to that of the kind type parameter of [ASCII character](#) if the processor supports such a kind; otherwise the result has the value -1 . If NAME has the value ISO_10646, then the result has a value equal to that of the kind type parameter of the [ISO 10646 character](#) kind (corresponding to UCS-4 as specified in ISO/IEC 10646) if the processor supports such a kind; otherwise the result has the value -1 . If NAME is a processor-defined name of some other character kind supported by the processor, then the result has a value equal to that kind type parameter value. If NAME is not the name of a supported character type, then the result has the value -1 . The NAME is interpreted without respect to case or trailing blanks.

Examples. SELECTED_CHAR_KIND ('ASCII') has the value 1 on a processor that uses 1 as the kind type parameter for the [ASCII character](#) set. The following subroutine produces a Japanese date stamp.

```

1      SUBROUTINE create_date_string(string)
2          INTRINSIC date_and_time,selected_char_kind
3          INTEGER,PARAMETER :: ucs4 = selected_char_kind("ISO_10646")
4          CHARACTER(1,UCS4),PARAMETER :: nen=CHAR(INT(Z'5e74'),UCS4), & !year
5              gatsu=CHAR(INT(Z'6708'),UCS4), & !month
6              nichu=CHAR(INT(Z'65e5'),UCS4) !day
7          CHARACTER(len= *, kind= ucs4) string
8          INTEGER values(8)
9          CALL date_and_time(values=values)
10         WRITE(string,1) values(1),nen,values(2),gatsu,values(3),nichu
11     1 FORMAT(I0,A,I0,A,I0,A)
12     END SUBROUTINE

```

16.9.181 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 $-10^R < n < 10^R$, or if no such kind type parameter is available on the processor, the result is -1 . If more than one kind type parameter meets the criterion, the value returned is the one with the smallest decimal exponent range, unless there are several such values, in which case the smallest of these kind values is returned.

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.

16.9.182 SELECTED_LOGICAL_KIND (BITS)

1 **Description.** Logical kind selection.

2 **Class.** [Transformational function](#).

3 **Argument.** BITS 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 a logical type whose storage size in bits is at least BITS, 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 storage size, unless there are several such values, in which case the smallest of these kind values is returned.

6 **Example.** Assume a processor supports four logical kinds with kind type parameter values 8, 16, 32, and 64 for representations with those storage sizes. On this processor, SELECTED_LOGICAL_KIND (1) has the value 8, SELECTED_LOGICAL_KIND (12) has the value 16, and SELECTED_LOGICAL_KIND (128) has the value -1 .

16.9.183 SELECTED_REAL_KIND ([P, R, RADIX])

Description. Real kind selection.

Class. Transformational function.

Arguments. At least one argument shall be present.

P (optional) shall be an integer scalar.

R (optional) shall be an integer scalar.

RADIX (optional) shall be an integer scalar.

Result Characteristics. Default integer scalar.

Result Value. If P or R is absent, the result value is the same as if it were present with the value zero. If RADIX is absent, there is no requirement on the radix of the selected kind.

The result has a value equal to a value of the kind type parameter of 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.

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.

If more than one kind type parameter value meets the criteria, the value returned is the one with the smallest decimal precision, unless there are several such values, in which case the smallest of these kind values is returned.

Example. 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.

16.9.184 SET_EXPONENT (X, I)

Description. Real value with specified exponent.

Class. Elemental function.

Arguments.

X shall be of type real.

I shall be of type integer.

Result Characteristics. Same as X.

Result Value. If X has the value zero, the result has the same value as X. If X is an IEEE infinity, the result is an IEEE NaN. If X is an IEEE NaN, the result is the same NaN. Otherwise, the result has the value $X \times b^{I-e}$, where b and e are as defined in 16.4 for the representation for the value of X in the extended real model for the kind of X.

1 6 **Example.** SET_EXPONENT (3.0, 1) has the value 1.5 for reals whose model is as in 16.4, NOTE.

2 16.9.185 SHAPE (SOURCE [, KIND])

3 1 **Description.** Shape of an array or a scalar.

4 2 **Class.** Inquiry function.

5 3 **Arguments.**

6 SOURCE may be of any type. It shall not be an unallocated allocatable variable or a pointer that is not
7 associated. It shall not be an assumed-size array.

8 KIND (optional) shall be a scalar integer constant expression.

9 4 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the value
10 of KIND; otherwise the kind type parameter is that of default integer type. The result is an array of rank
11 one whose size is equal to the rank of SOURCE.

12 5 **Result Value.** The result has a value whose i^{th} element is equal to the extent of dimension i of SOURCE,
13 except that if SOURCE is assumed-rank, and associated with an assumed-size array, the last element is equal
14 to -1 .

15 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
16 size zero.

17 16.9.186 SHIFTA (I, SHIFT)

18 1 **Description.** Right shift with fill.

19 2 **Class.** Elemental function.

20 3 **Arguments.**

21 I shall be of type integer.

22 SHIFT shall be of type integer. It shall be nonnegative and less than or equal to BIT_SIZE (I).

23 4 **Result Characteristics.** Same as I.

24 5 **Result Value.** The result has the value obtained by shifting the bits of I to the right SHIFT bits and replic-
25 ating the leftmost bit of I in the left SHIFT bits.

26 6 If SHIFT is zero the result is I. Bits shifted out from the right are lost. The model for the interpretation of
27 an integer value as a sequence of bits is in 16.3.

28 7 **Example.** SHIFTA (IBSET (0, BIT_SIZE (0) $-$ 1), 2) is equal to SHIFTL (7, BIT_SIZE (0) $-$ 3).

29 16.9.187 SHIFTL (I, SHIFT)

30 1 **Description.** Left shift.

31 2 **Class.** Elemental function.

32 3 **Arguments.**

33 I shall be of type integer.

34 SHIFT shall be of type integer. It shall be nonnegative and less than or equal to BIT_SIZE (I).

1 4 **Result Characteristics.** Same as I.

2 5 **Result Value.** The value of the result is ISHFT (I, SHIFT).

3 6 **Examples.** SHIFTL (3, 1) has the value 6.

4 16.9.188 SHIFTR (I, SHIFT)

5 1 **Description.** Right shift.

6 2 **Class.** Elemental function.

7 3 **Arguments.**

8 I shall be of type integer.

9 SHIFT shall be of type integer. It shall be nonnegative and less than or equal to BIT_SIZE (I).

10 4 **Result Characteristics.** Same as I.

11 5 **Result Value.** The value of the result is ISHFT (I, −SHIFT).

12 6 **Examples.** SHIFTR (3, 1) has the value 1.

13 16.9.189 SIGN (A, B)

14 1 **Description.** Magnitude of A with the sign of B.

15 2 **Class.** Elemental function.

16 3 **Arguments.**

17 A shall be of type integer or real.

18 B shall be of the same type as A.

19 4 **Result Characteristics.** Same as A.

20 5 **Result Value.**

21 *Case (i):* If $B > 0$, the value of the result is $|A|$.

22 *Case (ii):* If $B < 0$, the value of the result is $-|A|$.

23 *Case (iii):* If B is of type integer and $B=0$, the value of the result is $|A|$.

24 *Case (iv):* If B is of type real and is zero, then:

- 25 • if the processor does not distinguish between positive and negative real zero, or if B is
- 26 positive real zero, the value of the result is $|A|$;
- 27 • if the processor distinguishes between positive and negative real zero, and B is negative
- 28 real zero, the value of the result is $-|A|$.

29 6 **Example.** SIGN (−3.0, 2.0) has the value 3.0.

30 16.9.190 SIN (X)

31 1 **Description.** Sine function.

32 2 **Class.** Elemental function.

1 3 **Argument.** X shall be of type real or complex.

2 4 **Result Characteristics.** Same as X.

3 5 **Result Value.** The result has a value equal to a processor-dependent approximation to $\sin(X)$. If X is of type
4 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.

5 6 **Example.** $\text{SIN}(1.0)$ has the value 0.84147098 (approximately).

6 16.9.191 SIND (X)

7 1 **Description.** Degree sine function.

8 2 **Class.** [Elemental](#) function.

9 3 **Argument.** X shall be of type real.

10 4 **Result Characteristics.** Same as X.

11 5 **Result Value.** The result has a value equal to a processor-dependent approximation to the sine of X, which
12 is regarded as a value in degrees.

13 6 **Example.** $\text{SIND}(180.0)$ has the value 0.0 (approximately).

14 16.9.192 SINH (X)

15 1 **Description.** Hyperbolic sine function.

16 2 **Class.** [Elemental](#) function.

17 3 **Argument.** X shall be of type real or complex.

18 4 **Result Characteristics.** Same as X.

19 5 **Result Value.** The result has a value equal to a processor-dependent approximation to $\sinh(X)$. If X is of
20 type complex its imaginary part is regarded as a value in radians.

21 6 **Example.** $\text{SINH}(1.0)$ has the value 1.1752012 (approximately).

22 16.9.193 SINPI (X)

23 1 **Description.** Circular sine function.

24 2 **Class.** [Elemental](#) function.

25 3 **Argument.** X shall be of type real.

26 4 **Result Characteristics.** Same as X.

27 5 **Result Value.** The result has a value equal to a processor-dependent approximation to the sine of X, which
28 is regarded as a value in half-revolutions; thus, $\text{SINPI}(X)$ is approximately equal to [SIN](#) ($X \times \pi$).

29 6 **Example.** $\text{SINPI}(1.0)$ has the value 0.0 (approximately).

16.9.194 SIZE (ARRAY [, DIM, KIND])

Description. Size of an array or one extent.

Class. Inquiry function.

Arguments.

ARRAY shall be **assumed-rank** or an array. It shall not be an unallocated **allocatable** variable or a pointer that is not associated. 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**.

Result Characteristics. Integer scalar. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type.

Result Value. If DIM is present, the result has a value equal to the extent of dimension DIM of ARRAY, except that if ARRAY is **assumed-rank** and associated with an **assumed-size array** and DIM is present with a value equal to the rank of ARRAY, the value is -1 .

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.

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

If ARRAY is **assumed-rank** and has rank zero, DIM cannot be present since it cannot satisfy the requirement $1 \leq \text{DIM} \leq 0$.

16.9.195 SPACING (X)

Description. Spacing of model numbers.

Class. Elemental function.

Argument. X shall be of type real.

Result Characteristics. Same as X.

Result Value. If X does not have the value zero and is not an IEEE infinity or **NaN**, the result has the value b^{e-p} , where b , e , and p are as defined in 16.4 for the value nearest to X in the model for real values whose kind type parameter is that of X, provided this result is representable; otherwise, the result is the same as that of **TINY** (X). If there are two extended model values equally near to X, the value of greater absolute value is taken. If X has the value zero, the result is the same as that of **TINY** (X). If X is an IEEE infinity, the result is an **IEEE NaN**. If X is an **IEEE NaN**, the result is that **NaN**.

Example. SPACING (3.0) has the value 2^{-22} for reals whose model is as in 16.4, NOTE.

16.9.196 SPLIT (STRING, SET, POS [, BACK])

1 Description. Parse a string into tokens, one at a time.

2 Class. *Simple* subroutine.

3 Arguments.

STRING shall be a scalar of type character. It is an *INTENT (IN)* argument.

SET shall be a scalar of type character with the same kind type parameter as STRING. It is an *INTENT (IN)* argument. Each character in SET is a token delimiter. A sequence of zero or more characters in STRING delimited by any token delimiter, or the beginning or end of STRING, comprise a token. Thus, two consecutive token delimiters in STRING, or a token delimiter in the first or last character of STRING, indicate a token with zero length.

POS shall be an integer scalar. It is an *INTENT (INOUT)* argument. If BACK is present with the value true, the value of POS shall be in the range $0 < \text{POS} \leq \text{LEN}(\text{STRING}) + 1$; otherwise it shall be in the range $0 \leq \text{POS} \leq \text{LEN}(\text{STRING})$.

If BACK is absent or is present with the value false, POS is assigned the position of the leftmost token delimiter in STRING whose position is greater than POS, or if there is no such character, it is assigned a value one greater than the length of STRING. This identifies a token with starting position one greater than the value of POS on invocation, and ending position one less than the value of POS on return.

If BACK is present with the value true, POS is assigned the position of the rightmost token delimiter in STRING whose position is less than POS, or if there is no such character, it is assigned the value zero. This identifies a token with ending position one less than the value of POS on invocation, and starting position one greater than the value of POS on return.

If SPLIT is invoked with a value for POS in the range $1 \leq \text{POS} \leq \text{LEN}(\text{STRING})$, and the value of STRING (POS:POS) is not equal to any character in SET, the token identified by SPLIT will not comprise a complete token as described in the description of the SET argument, but rather a partial token.

BACK (optional) shall be a logical scalar. It is an *INTENT (IN)* argument.

4 Example.

Execution of

```
CHARACTER (LEN=:), ALLOCATABLE :: INPUT
CHARACTER (LEN=2) :: SET = ', '
INTEGER P
INPUT = "one,last example"
P = 0
DO
  IF (P > LEN (INPUT)) EXIT
  ISTART = P + 1
  CALL SPLIT (INPUT, SET, P)
  IEND = P - 1
  PRINT ' (T7,A) ', INPUT (ISTART:IEND)
END DO
```

will print

```
one
last
example
```

16.9.197 SPREAD (SOURCE, DIM, NCOPIES)

Description. Value replicated in a new dimension.

Class. Transformational function.

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.

Result Characteristics. The result is an array of the same type and type parameters as SOURCE and of **rank** $n + 1$, where n is the **rank** of SOURCE.

Case (i): If SOURCE is scalar, the shape of the result is $(\text{MAX}(\text{NCOPIES}, 0))$.

Case (ii): If SOURCE is an array with shape $[d_1, d_2, \dots, d_n]$, the shape of the result is $[d_1, d_2, \dots, d_{\text{DIM}-1}, \text{MAX}(\text{NCOPIES}, 0), d_{\text{DIM}}, \dots, d_n]$.

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 $(r_1, r_2, \dots, r_{n+1})$ has the value $\text{SOURCE}(r_1, r_2, \dots, r_{\text{DIM}-1}, r_{\text{DIM}+1}, \dots, r_{n+1})$.

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.

16.9.198 SQRT (X)

Description. Square root.

Class. Elemental function.

Argument. X shall be of type real or complex. If X is real, its value shall be greater than or equal to zero.

Result Characteristics. Same as X.

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.

Example. SQRT (4.0) has the value 2.0 (approximately).

16.9.199 STOPPED_IMAGES ([TEAM, KIND])

Description. Indices of stopped images.

Class. Transformational function.

Arguments.

TEAM (optional) shall be a scalar of type [TEAM_TYPE](#) from the intrinsic module [ISO_FORTRAN_ENV](#), whose value identifies the [current](#) or an [ancestor](#) team. If TEAM is absent the team specified is the [current team](#).

KIND (optional) shall be a scalar integer [constant expression](#).

4 Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default integer type. The result is an array of rank one whose size is equal to the number of images in the specified [team](#) that have initiated [normal termination](#).

5 Result Value. The elements of the result are the values of the indices of the images that are known to have initiated [normal termination](#) in the specified team, in numerically increasing order. If the executing image has previously executed an [image control statement](#) whose STAT= specifier assigned the value [STAT_STOPPED_IMAGE](#) from the intrinsic module [ISO_FORTRAN_ENV](#) or invoked a [collective subroutine](#) whose STAT argument was assigned [STAT_STOPPED_IMAGE](#), at least one of the images participating in that [image control statement](#) or collective invocation shall be known to have initiated [normal termination](#).

6 Examples. If image 3 is the only image in the [current team](#) that is known to have initiated [normal termination](#), STOPPED_IMAGES() will have the value [3]. If there are no images in the [current team](#) that have initiated [normal termination](#), the value of STOPPED_IMAGES() will be a zero-sized array.

16.9.200 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](#) (19.5.3) applies, the result is consistent with the [named constants](#) defined in the intrinsic module [ISO_FORTRAN_ENV](#).

NOTE 1

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 2

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

1 intrinsic module [ISO_FORTRAN_ENV](#).

2 16.9.201 SUM (ARRAY, DIM [, MASK]) or SUM (ARRAY [, MASK])

3 1 **Description.** Array reduced by addition.

4 2 **Class.** [Transformational function](#).

5 3 **Arguments.**

6 ARRAY shall be an array of [numeric type](#).

7 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.

8 MASK (optional) shall be of type logical and shall be [conformable](#) with ARRAY.

9 4 **Result Characteristics.** The result is of the same type and kind type parameter as ARRAY. It is scalar if
10 DIM does not appear; otherwise, the result has [rank](#) $n - 1$ and shape $[d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n]$ where
11 $[d_1, d_2, \dots, d_n]$ is the shape of ARRAY.

12 5 **Result Value.**

13 *Case (i):* The result of SUM (ARRAY) has a value equal to a processor-dependent approximation to the
14 sum of all the elements of ARRAY or has the value zero if ARRAY has size zero.

15 *Case (ii):* The result of SUM (ARRAY, MASK = MASK) has a value equal to a processor-dependent approx-
16 imation to the sum of the elements of ARRAY corresponding to the true elements of MASK or
17 has the value zero if there are no true elements.

18 *Case (iii):* If ARRAY has [rank](#) one, SUM (ARRAY, DIM = DIM [, MASK = MASK]) has a value equal to that
19 of SUM (ARRAY [, MASK = MASK]). Otherwise, the value of element $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1},$
20 $\dots, s_n)$ of SUM (ARRAY, DIM = DIM [, MASK = MASK]) is equal to

21
$$\text{SUM}(\text{ARRAY}(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n) \\ [, \text{MASK} = \text{MASK}(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)]).$$

22 6 **Examples.**

23 *Case (i):* The value of SUM ([1, 2, 3]) is 6.

24 *Case (ii):* SUM (C, MASK= C > 0.0) forms the sum of the positive elements of C.

25 *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].

26 16.9.202 SYSTEM_CLOCK ([COUNT, COUNT_RATE, COUNT_MAX])

27 1 **Description.** Query system clock.

28 2 **Class.** Subroutine.

29 3 **Arguments.**

30 COUNT (optional) shall be an integer scalar with a decimal exponent range no smaller than that of default
31 integer. It is an [INTENT \(OUT\)](#) argument. It is assigned a processor-dependent value based
32 on the value of a processor clock, or –HUGE (COUNT) if there is no clock for the invoking
33 [image](#). The processor-dependent value is incremented by one for each clock count until the
34 value COUNT_MAX is reached and is reset to zero at the next count. It lies in the range 0 to
35 COUNT_MAX if there is a clock.

COUNT_RATE (optional) shall be an integer or real scalar. If it is of type integer, it shall have a decimal exponent range no smaller than that of default integer. 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 with a decimal exponent range no smaller than that of default integer. 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 In a reference to SYSTEM_CLOCK, all integer arguments shall have the same kind type parameter.

5 Whether an **image** has no clock, has one or more clocks of its own, or shares a clock with another **image**, is processor dependent.

6 If more than one clock is available, the types and kinds of the arguments to SYSTEM_CLOCK determine which clock is accessed. The processor should document the relationship between the clock selection and the argument characteristics.

7 Different invocations of SYSTEM_CLOCK should use the same types and kinds for the arguments, to ensure that any timing calculations are based on the same clock.

8 It is recommended that all references to SYSTEM_CLOCK use integer arguments with a decimal exponent range of at least 18. This lets the processor select the most accurate clock available while minimizing how often the COUNT value resets to zero.

9 **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 $M = 24 \times 3600 \times 18.20648193 - 1 = 1573039$.

16.9.203 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).

16.9.204 TAND (X)

1 **Description.** Degree tangent function.

2 **Class.** **Elemental** function.

3 **Argument.** X shall be of type real.

1 4 **Result Characteristics.** Same as X.

2 5 **Result Value.** The result has a value equal to a processor-dependent approximation to the tangent of X,
3 which is regarded as a value in degrees.

4 6 **Example.** TAND (180.0) has the value 0.0 (approximately).

5 16.9.205 TANH (X)

6 1 **Description.** Hyperbolic tangent function.

7 2 **Class.** [Elemental](#) function.

8 3 **Argument.** X shall be of type real or complex.

9 4 **Result Characteristics.** Same as X.

10 5 **Result Value.** The result has a value equal to a processor-dependent approximation to $\tanh(X)$. If X is of
11 type complex its imaginary part is regarded as a value in radians.

12 6 **Example.** TANH (1.0) has the value 0.76159416 (approximately).

13 16.9.206 TANPI (X)

14 1 **Description.** Circular tangent function.

15 2 **Class.** [Elemental](#) function.

16 3 **Argument.** X shall be of type real.

17 4 **Result Characteristics.** Same as X.

18 5 **Result Value.** The result has a value equal to a processor-dependent approximation to the tangent of X,
19 which is regarded as a value in half-revolutions; thus, TANPI (X) is approximately equal to [TAN](#) ($X \times \pi$).

20 6 **Example.** TANPI (1.0) has the value 0.0 (approximately).

21 16.9.207 TEAM_NUMBER ([TEAM])

22 1 **Description.** Team number.

23 2 **Class.** [Transformational](#) function.

24 3 **Argument.** TEAM (optional) shall be a scalar of type [TEAM_TYPE](#) from the intrinsic module [ISO_FOR-](#)
25 [TRAN_ENV](#), whose value identifies the [current](#) or an [ancestor](#) team. If TEAM is absent, the team specified
26 is the [current team](#).

27 4 **Result Characteristics.** Default integer scalar.

28 5 **Result Value.** The result has the value -1 if the specified team is the [initial team](#); otherwise, the result
29 value is equal to the positive integer that identifies the specified team among its [sibling teams](#).

30 6 **Example.** The team number can be used to control which statements get executed, for example:

```
31      TYPE (TEAM_TYPE) :: ODD_EVEN
32      ...
33      FORM TEAM (2-MOD (ME,2), ODD_EVEN)
```

```

1      ...
2      CHANGE TEAM (ODD_EVEN)
3      SELECT CASE (TEAM_NUMBER())
4      CASE (1)
5          ! Case for images with odd image indices in the parent team.
6      CASE (2)
7          ! Case for images with even image indices in the parent team.
8      END SELECT
9      END TEAM

```

16.9.208 THIS_IMAGE ([TEAM]) or THIS_IMAGE (COARRAY [, TEAM]) or THIS_IMAGE (COARRAY, DIM [, TEAM])

1 **Description.** Cosubscript(s) for this image.

2 **Class.** Transformational function.

3 **Arguments.**

COARRAY shall be a *coarray* of any type. If it is *allocatable* it shall be allocated. If its *designator* has more than one *part-ref*, the rightmost *part-ref* shall have nonzero *corank*.

DIM shall be an integer scalar. Its value shall be in the range $1 \leq \text{DIM} \leq n$, where n is the *corank* of COARRAY.

TEAM (optional) shall be a scalar of type *TEAM_TYPE* from the intrinsic module *ISO_FORTRAN_ENV*, whose value identifies the *current* or an *ancestor team*. If COARRAY appears, it shall be established in that *team*.

4 **Result Characteristics.** Default integer. It is scalar if COARRAY does not appear or DIM appears; otherwise, the result has *rank* one and its size is equal to the *corank* of COARRAY.

5 **Result Value.**

Case (i): The result of THIS_IMAGE ([TEAM]) is a scalar with a value equal to the *index* of the invoking image in the *team* specified by TEAM, if present, or in the *current team* if absent.

Case (ii): The result of THIS_IMAGE (COARRAY [, TEAM = TEAM]) is the sequence of *cosubscript* values for COARRAY that would specify the invoking image in the *team* specified by TEAM, if present, or in the *current team* if absent.

Case (iii): The result of THIS_IMAGE (COARRAY, DIM [, TEAM = TEAM]) is the value of *cosubscript* DIM in the sequence of *cosubscript* values for COARRAY that would specify the invoking image in the *team* specified by TEAM, if present, or in the *current team* if absent.

6 **Examples.**

Case (i): 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].

Case (ii): 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]

```

16.9.209 TINY (X)

Description. Smallest positive model number.

Class. [Inquiry function](#).

Argument. X shall be a real scalar or array.

Result Characteristics. Scalar with the same type and kind type parameter as X.

Result Value. The result has the value $b^{e_{\min}-1}$ where b and $e_{\min}$ are as defined in [16.4](#) for the model representing numbers of the same type and kind type parameter as X.

Example. TINY (X) has the value 2^{-127} for real X whose model is as in [16.4](#), [NOTE](#).

16.9.210 TOKENIZE (STRING, SET, TOKENS [, SEPARATOR]) or TOKENIZE (STRING, SET, FIRST, LAST)

Description. Parse a string into tokens.

Class. [Simple](#) subroutine.

Arguments.

STRING shall be a scalar of type character. It is an [INTENT \(IN\)](#) argument.

SET shall be a scalar of type character with the same kind type parameter as STRING. It is an [INTENT \(IN\)](#) argument. Each character in SET is a token delimiter. A sequence of zero or more characters in STRING delimited by any token delimiter, or the beginning or end of STRING, comprise a token. Thus, two consecutive token delimiters in STRING, or a token delimiter in the first or last character of STRING, indicate a token with zero length.

TOKENS shall be of type character with the same kind type parameter as STRING. It is an [INTENT \(OUT\)](#) argument. It shall not be a [coarray](#) or a [coindexed object](#). It shall be an allocatable array of rank one with deferred length. It is allocated with the lower bound equal to one and the upper bound equal to the number of tokens in STRING, and with character length equal to the length of the longest token.

The tokens in STRING are assigned in the order found, as if by [intrinsic assignment](#), to the elements of TOKENS, in array element order.

SEPARATOR (optional) shall be of type character with the same kind type parameter as STRING. It is an [INTENT \(OUT\)](#) argument. It shall not be a [coarray](#) or a [coindexed object](#). It shall be an allocatable array of rank one with deferred length. It is allocated with the lower bound equal to one and the upper bound equal to one less than the number of tokens in STRING, and with character length equal to one. Each element SEPARATOR(i) is assigned the value of the i^{th} token delimiter in STRING.

FIRST shall be an allocatable array of type integer and rank one. It is an [INTENT \(OUT\)](#) argument. It shall not be a [coarray](#) or a [coindexed object](#). It is allocated with the lower bound equal to one and the upper bound equal to the number of tokens in STRING. Each element is assigned, in array element order, the starting position of each token in STRING, in the order found. If a token has zero length, the starting position is equal to one if the token is at the beginning of STRING, and one greater than the position of the preceding delimiter otherwise.

LAST shall be an allocatable array of type integer and rank one. It is an [INTENT \(OUT\)](#) argument. It shall not be a [coarray](#) or a [coindexed object](#). It is allocated with the lower bound equal to

one and the upper bound equal to the number of tokens in STRING. Each element is assigned, in array element order, the ending position of each token in STRING, in the order found. If a token has zero length, the ending position is one less than the starting position.

4 Examples.

Execution of

```
CHARACTER (LEN=:), ALLOCATABLE :: STRING
CHARACTER (LEN=:), ALLOCATABLE, DIMENSION (:) :: TOKENS
CHARACTER (LEN=2) :: SET = ','
STRING = 'first,second,third'
CALL TOKENIZE (STRING, SET, TOKENS)
```

will assign the value ['first ', 'second', 'third '] to TOKENS.

Execution of

```
CHARACTER (LEN=:), ALLOCATABLE :: STRING
CHARACTER (LEN=2) :: SET = ','
INTEGER, DIMENSION (:) , ALLOCATABLE :: FIRST, LAST
STRING = 'first,second,,fourth'
CALL TOKENIZE (STRING, SET, FIRST, LAST)
```

will assign the value [1, 7, 14, 15] to FIRST, and the value [5, 12, 13, 20] to LAST.

16.9.211 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 [16.3](#).

6 **Examples.** TRAILZ (8) has the value 3.

16.9.212 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. If the storage size of SOURCE is greater than zero and MOLD is an array, a scalar with the type and [type parameters](#) of MOLD shall not have a storage size equal to zero.

SIZE (optional) shall be an integer scalar. The corresponding [actual argument](#) shall not be an optional dummy argument.

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.

1 *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
2 as possible such that its physical representation is not shorter than that of SOURCE.

3 *Case (iii):* If SIZE is present, the result is an array of **rank** one and size SIZE.

4 5 **Result Value.** If the physical representation of the result has the same length as that of SOURCE, the phys-
5 ical representation of the result is that of SOURCE. If the physical representation of the result is longer than
6 that of SOURCE, the physical representation of the leading part is that of SOURCE and the remainder is pro-
7 cessor dependent. If the physical representation of the result is shorter than that of SOURCE, the physical
8 representation of the result is the leading part of SOURCE. If D and E are scalar variables such that the phys-
9 ical representation of D is as long as or longer than that of E, the value of TRANSFER (TRANSFER (E, D), E)
10 shall be the value of E. IF D is an array and E is an array of **rank** one, the value of TRANSFER (TRANS-
11 FER (E, D), E, SIZE (E)) shall be the value of E.

12 6 **Examples.**

13 *Case (i):* TRANSFER (1082130432, 0.0) has the value 4.0 on a processor that represents the values 4.0
14 and 1082130432 as the string of binary digits 0100 0000 1000 0000 0000 0000 0000 0000.

15 *Case (ii):* TRANSFER ([1.1, 2.2, 3.3], [(0.0, 0.0)]) is a complex rank-one array of length two whose first
16 element has the value (1.1, 2.2) and whose second element has a real part with the value 3.3.
17 The imaginary part of the second element is processor dependent.

18 *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
19 element has the value (1.1, 2.2).

20 **16.9.213 TRANSPOSE (MATRIX)**

21 1 **Description.** Transpose of an array of **rank** two.

22 2 **Class.** Transformational function.

23 3 **Argument.** MATRIX shall be a rank-two array of any type.

24 4 **Result Characteristics.** The result is an array of the same type and type parameters as MATRIX and with
25 **rank** two and shape $[n, m]$ where $[m, n]$ is the shape of MATRIX.

26 5 **Result Value.** Element (i, j) of the result has the value $\text{MATRIX}(j + \text{LBOUND}(\text{MATRIX}, 1) - 1, i + \text{LBOUND}$
27 $(\text{MATRIX}, 2) - 1)$.

28 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 $\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$.

29 **16.9.214 TRIM (STRING)**

30 1 **Description.** String without trailing blanks.

31 2 **Class.** Transformational function.

32 3 **Argument.** STRING shall be a character scalar.

33 4 **Result Characteristics.** Character with the same kind type parameter value as STRING and with a length
34 that is the length of STRING less the number of trailing blanks in STRING. If STRING contains no nonblank
35 characters, the result has zero length.

36 5 **Result Value.** The value of the result is the same as STRING except any trailing blanks are removed.

1 6 **Example.** TRIM (' A B ') has the value ' A B '.

2 16.9.215 UBOUND (ARRAY [, DIM, KIND])

3 1 **Description.** Upper bound(s).

4 2 **Class.** Inquiry function.

5 3 **Arguments.**

6 ARRAY shall be **assumed-rank** or an array. It shall not be an unallocated **allocatable** array or a pointer
7 that is not associated. If ARRAY is an **assumed-size array**, DIM shall be present with a value
8 less than the **rank** of ARRAY.

9 DIM (optional) shall be an integer scalar with a value in the range $1 \leq \text{DIM} \leq n$, where n is the **rank**
10 of ARRAY. The corresponding **actual argument** shall not be an optional dummy argument, a
11 disassociated pointer, or an unallocated allocatable.

12 KIND (optional) shall be a scalar integer **constant expression**.

13 4 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the value
14 of KIND; otherwise the kind type parameter is that of default integer type. The result is scalar if DIM is
15 present; otherwise, the result is an array of **rank** one and size n , where n is the **rank** of ARRAY.

16 5 **Result Value.**

17 *Case (i):* If DIM is present, ARRAY is a **whole array**, and dimension DIM of ARRAY has nonzero extent,
18 the result has a value equal to the upper bound for subscript DIM of ARRAY. Otherwise, if DIM is
19 present and ARRAY is **assumed-rank**, the value of the result is as if ARRAY were a **whole array**,
20 with the extent of the final dimension of ARRAY when ARRAY is associated with an **assumed-**
21 **size array** being considered to be -1 . Otherwise, if DIM is present, the result has a value equal
22 to the number of elements in dimension DIM of ARRAY.

23 *Case (ii):* If ARRAY has rank zero, UBOUND (ARRAY) has a value that is a zero-sized array. Otherwise,
24 UBOUND (ARRAY) has a value whose i^{th} element is equal to UBOUND (ARRAY, i), for $i = 1, 2,$
25 $\dots, n$, where n is the **rank** of ARRAY. UBOUND (ARRAY, KIND=KIND) has a value whose i^{th}
26 element is equal to UBOUND (ARRAY, i , KIND=KIND), for $i = 1, 2, \dots, n$, where n is the **rank**
27 of ARRAY.

28 6 **Examples.** If A is declared by the statement

29 REAL A (2:3, 7:10)

30 then UBOUND (A) is [3, 10] and UBOUND (A, DIM = 2) is 10.

NOTE

If ARRAY is **assumed-rank** and has rank zero, DIM cannot be present since it cannot satisfy the requirement $1 \leq \text{DIM} \leq 0$.

31 16.9.216 UCBOUND (COARRAY [, DIM, KIND])

32 1 **Description.** Upper **cobound**(s) of a **coarray**.

33 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. If its **designator** has more than one **part-ref**, the rightmost **part-ref** shall have nonzero **corank**.

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** whose **index** is equal to the number of **images** in the **current team**.

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 $i = 1, 2, \dots, n$, 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**.

16.9.217 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 $i = 1, 2, \dots, t$, 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

1 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
2 true and “.” represents false, then the result of UNPACK (V, MASK = Q, FIELD = M) has the value $\begin{bmatrix} 1 & 2 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 3 \end{bmatrix}$
3 and the result of UNPACK (V, MASK = Q, FIELD = 0) has the value $\begin{bmatrix} 0 & 2 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 3 \end{bmatrix}$.

4 **16.9.218 VERIFY (STRING, SET [, BACK, KIND])**

5 1 **Description.** Character set non-membership search.

6 2 **Class.** [Elemental](#) function.

7 3 **Arguments.**

8 STRING shall be of type character.

9 SET shall be of type character with the same kind type parameter as STRING.

10 BACK (optional) shall be of type logical.

11 KIND (optional) shall be a scalar integer [constant expression](#).

12 4 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the value
13 of KIND; otherwise the kind type parameter is that of default integer type.

14 5 **Result Value.**

15 *Case (i):* If BACK is absent or has the value false and if STRING contains at least one character that is
16 not in SET, the value of the result is the position of the leftmost character of STRING that is not
17 in SET.

18 *Case (ii):* If BACK is present with the value true and if STRING contains at least one character that is not
19 in SET, the value of the result is the position of the rightmost character of STRING that is not
20 in SET.

21 *Case (iii):* The value of the result is zero if each character in STRING is in SET or if STRING has zero length.

22 6 **Examples.**

23 *Case (i):* VERIFY ('ABBA', 'A') has the value 2.

24 *Case (ii):* VERIFY ('ABBA', 'A', BACK = [.TRUE.](#)) has the value 3.

25 *Case (iii):* VERIFY ('ABBA', 'AB') has the value 0.

26 **16.10 Standard intrinsic modules**

27 **16.10.1 General**

28 1 This document defines five standard intrinsic modules: a Fortran environment module, a set of three mod-
29 ules to support floating-point exceptions and IEEE arithmetic, and a module to support interoperability
30 with the C programming language.

1 2 The intrinsic modules IEEE_EXCEPTIONS, IEEE_ARITHMETIC, and IEEE_FEATURES are described in Clause
 2 17. The intrinsic module ISO_C_BINDING is described in Clause 18. The module procedures described in
 3 16.10.2 are simple.

NOTE

The types and procedures defined in standard intrinsic modules are not themselves intrinsic.

4 3 A processor may extend the standard intrinsic modules to provide public entities in them in addition to
 5 those specified in this document.

6 16.10.2 The ISO_FORTRAN_ENV intrinsic module

7 16.10.2.1 General

8 1 The intrinsic module ISO_FORTRAN_ENV provides public entities relating to the Fortran environment.
 9 2 The processor shall provide the named constants, derived types, and procedures described in 16.10.2. In
 10 the detailed descriptions below, procedure names are generic and not specific.

11 16.10.2.2 ATOMIC_INT_KIND

12 1 The value of the default integer scalar constant ATOMIC_INT_KIND is the kind type parameter value of type
 13 integer variables for which the processor supports atomic operations specified by atomic subroutines.

14 16.10.2.3 ATOMIC_LOGICAL_KIND

15 1 The value of the default integer scalar constant ATOMIC_LOGICAL_KIND is the kind type parameter value of
 16 type logical variables for which the processor supports atomic operations specified by atomic subroutines.

17 16.10.2.4 CHARACTER_KINDS

18 1 The values of the elements of the default integer array constant CHARACTER_KINDS are the kind values
 19 supported by the processor for variables of type character. The order of the values is processor dependent.
 20 The rank of the array is one, its lower bound is one, and its size is the number of character kinds supported.

21 16.10.2.5 CHARACTER_STORAGE_SIZE

22 1 The value of the default integer scalar constant CHARACTER_STORAGE_SIZE is the size expressed in bits of
 23 the character storage unit (19.5.3.2).

24 16.10.2.6 COMPILER_OPTIONS ()

25 1 **Description.** Processor-dependent string describing the options that controlled the program translation
 26 phase.

27 2 **Class.** Transformational function.

28 3 **Argument.** None.

29 4 **Result Characteristics.** Default character scalar with processor-dependent length.

1 5 **Result Value.** A processor-dependent value which describes the options that controlled the translation
 2 phase of program execution. This value should include relevant information that could be useful for dia-
 3 gnosing problems at a later date.

4 6 **Example.** COMPILER_OPTIONS () might have the value '/OPTIMIZE /FLOAT=IEEE'.

5 16.10.2.7 COMPILER_VERSION ()

6 1 **Description.** Processor-dependent string identifying the program translation phase.

7 2 **Class.** Transformational function.

8 3 **Argument.** None.

9 4 **Result Characteristics.** Default character scalar with processor-dependent length.

10 5 **Result Value.** A processor-dependent value that identifies the name and version of the program translation
 11 phase of the processor. This value should include relevant information that could be useful for diagnosing
 12 problems at a later date.

13 6 **Example.** COMPILER_VERSION () might have the value 'Fast KL-10 Compiler Version 7'.

NOTE

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.

14 16.10.2.8 CURRENT_TEAM

15 1 The value of the default integer scalar constant CURRENT_TEAM identifies the [current team](#) when it is used
 16 as the LEVEL argument to [GET_TEAM](#).

17 16.10.2.9 ERROR_UNIT

18 1 The value of the default integer scalar constant ERROR_UNIT identifies the processor-dependent [precon-](#)
 19 [nected external unit](#) used for the purpose of error reporting (12.5). This [unit](#) may be the same as OUTPUT_
 20 UNIT. The value shall not be −1.

21 16.10.2.10 EVENT_TYPE

22 1 EVENT_TYPE is a derived type with private components. It is an extensible type with no type parameters.
 23 Each nonallocatable component is fully default-initialized.

24 2 A scalar variable of type EVENT_TYPE is an [event variable](#). The value of an event variable includes its event
 25 count, which is updated by execution of a sequence of [EVENT POST](#) or [EVENT WAIT](#) statements. The effect
 26 of each change is as if the intrinsic subroutine [ATOMIC_ADD](#) were executed with a variable that stores the
 27 event count as its ATOM argument. A coarray that is of type EVENT_TYPE may be referenced or defined
 28 during execution of a [segment](#) that is unordered relative to the execution of another [segment](#) in which
 29 that coarray is defined. The event count is of type integer with kind [ATOMIC_INT_KIND](#) from the intrinsic
 30 module [ISO_FORTRAN_ENV](#). The initial value of the event count of an event variable is zero.

- 1 C1603 A named entity with declared type EVENT_TYPE, or which has a noncoarray [potential subobject](#)
 2 [component](#) with declared type EVENT_TYPE, shall be a variable. A component that is of such a type
 3 shall be a data component.
- 4 C1604 A named variable with [declared type](#) EVENT_TYPE shall be a coarray. A named variable with a
 5 noncoarray [potential subobject component](#) of type EVENT_TYPE shall be a coarray.
- 6 C1605 An [event variable](#) shall not appear in a variable definition context except as the [event-variable](#) in an
 7 [EVENT POST](#) or [EVENT WAIT](#) statement, as an [allocate-object](#), or as an [actual argument](#) in a refer-
 8 ence to a procedure with an [explicit interface](#) if the corresponding [dummy argument](#) has [INTENT](#)
 9 [\(INOUT\)](#).
- 10 C1606 A variable with a nonpointer subobject of type EVENT_TYPE shall not appear in a variable defini-
 11 tion context except as an [allocate-object](#) in an [ALLOCATE statement](#) without a [SOURCE= specifier](#),
 12 as an [allocate-object](#) in a [DEALLOCATE statement](#), or as an [actual argument](#) in a reference to a pro-
 13 cedure with an [explicit interface](#) if the corresponding [dummy argument](#) has [INTENT \(INOUT\)](#).

NOTE 1

The restrictions against changing an [event variable](#) except via [EVENT POST](#) and [EVENT WAIT](#) statements ensure the integrity of its value and facilitate efficient implementation, particularly when special synchronization is needed for correct event handling.

NOTE 2

Updates to variables via atomic subroutines are coherent but not necessarily consistent, so a processor might have to use extra synchronization to obtain the consistency required for the [segments](#) ordered by [EVENT POST](#) and [EVENT WAIT](#) statements.

16.10.2.11 FILE_STORAGE_SIZE

- 1 The value of the default integer scalar constant FILE_STORAGE_SIZE is the size expressed in bits of the [file](#)
 2 [storage unit](#) ([12.3.5](#)).

16.10.2.12 INITIAL_TEAM

- 1 The value of the default integer scalar constant INITIAL_TEAM identifies the [initial team](#) when it is used as
 2 the LEVEL argument to [GET_TEAM](#).

16.10.2.13 INPUT_UNIT

- 1 The value of the default integer scalar constant INPUT_UNIT identifies the same processor-dependent [ex-](#)
 2 [ternal unit](#) as the one identified by an asterisk in a [READ statement](#); this [unit](#) is the one used for a [READ](#)
 3 [statement](#) that does not contain an input/output control list ([12.6.4.3](#)). This unit is [preconnected](#) for se-
 4 quential formatted input on [image](#) one in the [initial team](#) only, and is not [preconnected](#) on any other [image](#).
 5 The value shall not be -1 .

16.10.2.14 INT8, INT16, INT32, and INT64

- 1 The values of these default integer scalar [named constants](#) shall be those of the [kind type parameters](#) that
 2 specify an INTEGER type whose storage size expressed in bits is 8, 16, 32, and 64 respectively. If, for any of
 3 these constants, the processor supports more than one kind of that size, it is processor dependent which

1 kind value is provided. If the processor supports no kind of a particular size, that constant shall be equal
2 to -2 if the processor supports a kind with larger size and -1 otherwise.

3 **16.10.2.15 INTEGER_KINDS**

4 1 The values of the elements of the default integer array constant `INTEGER_KINDS` are the kind values sup-
5 ported by the processor for variables of type integer. The order of the values is processor dependent. The
6 rank of the array is one, its lower bound is one, and its size is the number of integer kinds supported.

7 **16.10.2.16 IOSTAT_END**

8 1 The value of the default integer scalar constant `IOSTAT_END` is assigned to the variable specified in an
9 `IOSTAT= specifier` (12.11.5) if an end-of-file condition occurs during execution of an `input statement` and
10 no error condition occurs. This value shall be negative.

11 **16.10.2.17 IOSTAT_EOR**

12 1 The value of the default integer scalar constant `IOSTAT_EOR` is assigned to the variable specified in an
13 `IOSTAT= specifier` (12.11.5) if an end-of-record condition occurs during execution of an `input statement`
14 and no end-of-file or error condition occurs. This value shall be negative and different from the value of
15 `IOSTAT_END`.

16 **16.10.2.18 IOSTAT_INQUIRE_INTERNAL_UNIT**

17 1 The value of the default integer scalar constant `IOSTAT_INQUIRE_INTERNAL_UNIT` is assigned to the vari-
18 able specified in an `IOSTAT= specifier` in an `INQUIRE statement` (12.10) if a *file-unit-number* identifies an
19 `internal unit` in that statement.

NOTE

This can only occur when a `defined input/output` procedure is called by the processor as the result of executing a
parent `data transfer statement` (12.6.4.8.3) for an `internal unit`.

20 **16.10.2.19 LOCK_TYPE**

21 1 `LOCK_TYPE` is a `derived type` with private components; no component is allocatable or a pointer. It is an
22 `extensible type` with no `type parameters`. All components have default initialization.

23 2 A scalar variable of type `LOCK_TYPE` is a `lock variable`. A lock variable can have one of two states: locked
24 and unlocked. The unlocked state is represented by the one value that is the default value of a `LOCK_-`
25 `TYPE` variable; this is the value specified by the `structure constructor` `LOCK_TYPE ( )`. The locked state is
26 represented by all other values. The value of a lock variable can be changed with the `LOCK` and `UNLOCK`
27 statements (11.7.10).

28 C1607 A named entity with declared type `LOCK_TYPE`, or which has a noncoarray `potential subobject`
29 `component` with declared type `LOCK_TYPE`, shall be a variable. A component that is of such a type
30 shall be a data component.

31 C1608 A named variable with `declared type` `LOCK_TYPE` shall be a `coarray`. A named variable with a non-
32 coarray `potential subobject component` of type `LOCK_TYPE` shall be a `coarray`.

C1609 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)**.

C1610 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

The restrictions against changing a lock variable except via the **LOCK** and **UNLOCK** statements ensure the integrity of its value and facilitate efficient implementation, particularly when special synchronization is needed for correct lock operation.

16.10.2.20 LOGICAL_KINDS

- 1 The values of the elements of the default integer array constant **LOGICAL_KINDS** are the kind values supported by the processor for variables of type logical. The order of the values is processor dependent. The rank of the array is one, its lower bound is one, and its size is the number of logical kinds supported.

16.10.2.21 LOGICAL8, LOGICAL16, LOGICAL32, and LOGICAL64

- 1 The values of these default integer scalar *named constants* shall be those of the *kind type parameters* that specify a **LOGICAL** type whose storage size expressed in bits is 8, 16, 32, and 64 respectively. If, for any of these constants, the processor supports more than one kind of that size, it is processor dependent which kind value is provided. If the processor supports no kind of a particular size, that constant shall be equal to -2 if the processor supports a kind with larger size and -1 otherwise.

16.10.2.22 NOTIFY_TYPE

- 1 **NOTIFY_TYPE** is a *derived type* with private components. It is an *extensible type* with no *type parameters*. Each nonallocatable component is fully default-initialized.
- 2 A scalar variable of type **NOTIFY_TYPE** is a *notify variable*. The value of a *notify variable* includes its notify count, which is updated by execution of *assignment statements* that have a **NOTIFY= specifier** and **NOTIFY WAIT statements**.
- 3 The effect of each update is as if the intrinsic subroutine **ATOMIC_ADD** were executed with a variable that stores the notify count as its **ATOM** argument. A coarray that is of type **NOTIFY_TYPE** may be referenced or defined during execution of a *segment* that is unordered relative to the execution of another *segment* in which that coarray is defined. The notify count is of type integer with kind **ATOMIC_INT_KIND** from the intrinsic module **ISO_FORTRAN_ENV**. The initial value of the notify count of a notify variable is zero.

C1611 A named entity with declared type **NOTIFY_TYPE**, or which has a noncoarray *potential subobject component* with declared type **NOTIFY_TYPE**, shall be a variable. A component that is of such a type shall be a data component.

C1612 A named variable with declared type **NOTIFY_TYPE** shall be a coarray. A named variable with a noncoarray *potential subobject component* of type **NOTIFY_TYPE** shall be a coarray.

C1613 A *notify variable* shall not appear in a variable definition context except as the *notify-variable* of a **NOTIFY= specifier** or **NOTIFY WAIT statement**, as an *allocate-object*, or as an *actual argument*

in a reference to a procedure with an **explicit interface** if the corresponding **dummy argument** has **INTENT (INOUT)**.

C1614 A variable with a nonpointer subobject of type NOTIFY_TYPE shall not appear in a variable definition context except as an *allocate-object* in an **ALLOCATE statement** without a **SOURCE= specifier**, as an *allocate-object* in a **DEALLOCATE statement**, or as an **actual argument** in a reference to a procedure with an **explicit interface** if the corresponding **dummy argument** has **INTENT (INOUT)**.

NOTE

The restrictions on changing a notify variable ensure the integrity of its value and facilitate efficient implementation, particularly when special synchronization is needed for correct notify handling.

16.10.2.23 NUMERIC_STORAGE_SIZE

- 1 The value of the default integer scalar constant NUMERIC_STORAGE_SIZE is the size expressed in bits of the **numeric storage unit** (19.5.3.2).

16.10.2.24 OUTPUT_UNIT

- 1 The value of the default integer scalar constant OUTPUT_UNIT identifies the same processor-dependent **external unit preconnected** for sequential formatted output as the one identified by an asterisk in a **WRITE statement** (12.6.4.3); this **unit** is the one used by a **PRINT statement**. The value shall not be -1 .

16.10.2.25 PARENT_TEAM

- 1 The value of the default integer scalar constant PARENT_TEAM identifies the **parent team** when it is used as the LEVEL argument to **GET_TEAM**.

16.10.2.26 REAL_KINDS

- 1 The values of the elements of the default integer array constant REAL_KINDS are the kind values supported by the processor for variables of type real. The order of the values is processor dependent. The rank of the array is one, its lower bound is one, and its size is the number of real kinds supported.

16.10.2.27 REAL16, 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 16, 32, 64, and 128 respectively. If, for any of these constants, the processor supports more than one kind of that size, it is processor dependent which kind value is provided. If the processor supports no kind of a particular size, that constant shall be equal to -2 if the processor supports kinds of a larger size and -1 otherwise.

16.10.2.28 STAT_FAILED_IMAGE

- 1 If the processor has the ability to detect that an image has failed, the value of the default integer scalar constant STAT_FAILED_IMAGE is positive; otherwise, the value of STAT_FAILED_IMAGE is negative. If an image involved in execution of an **image control statement**, a reference to a **coindexed object**, or execution of a **collective** or **atomic** subroutine has failed, and no other error condition occurs, the value of STAT_FAILED_IMAGE is assigned to the variable specified in a **STAT= specifier** in the execution of an **image control statement** or reference to a **coindexed object**, or to the STAT argument in an invocation of a **collective** or **atomic** subroutine.

16.10.2.29 STAT_LOCKED

The value of the default integer scalar constant STAT_LOCKED is assigned to the variable specified in a **STAT= specifier** (11.7.11) of a **LOCK statement** if the **lock variable** is locked by the executing **image**.

16.10.2.30 STAT_LOCKED_OTHER_IMAGE

The value of the default integer scalar constant STAT_LOCKED_OTHER_IMAGE is assigned to the variable specified in a **STAT= specifier** (11.7.11) of an **UNLOCK statement** if the **lock variable** is locked by another **image**.

16.10.2.31 STAT_STOPPED_IMAGE

The value of the default integer scalar constant STAT_STOPPED_IMAGE is assigned to the variable specified in a **STAT= specifier** (9.7.4, 11.7.11), if execution of the statement with that specifier requires synchronization with an image that has initiated **normal termination**. It is assigned to a STAT argument in a reference to a **collective subroutine** if any **image** of the **current team** has initiated **normal termination**. This value shall be positive.

16.10.2.32 STAT_UNLOCKED

The value of the default integer scalar constant STAT_UNLOCKED is assigned to the variable specified in a **STAT= specifier** (11.7.11) of an **UNLOCK statement** if the **lock variable** is unlocked.

16.10.2.33 STAT_UNLOCKED_FAILED_IMAGE

The value of the default integer scalar constant STAT_UNLOCKED_FAILED_IMAGE is assigned to the variable specified in a **STAT= specifier** (11.7.11) of a **LOCK statement** if the **lock variable** is unlocked because of the failure of the **image** that locked it.

16.10.2.34 TEAM_TYPE

TEAM_TYPE is a **derived type** with private components. It is an **extensible type** with no **type parameters**. Each nonallocatable component is fully default-initialized.

A scalar variable of type TEAM_TYPE is a **team variable**, and can identify a team. The default initial value of a team variable does not identify any team.

16.10.2.35 Uniqueness of named constant values

The values of these **named constants** shall be distinct:

IOSTAT_INQUIRE_INTERNAL_UNIT	STAT_STOPPED_IMAGE
STAT_FAILED_IMAGE	STAT_UNLOCKED
STAT_LOCKED	STAT_UNLOCKED_FAILED_IMAGE
STAT_LOCKED_OTHER_IMAGE	

17 Exceptions and IEEE arithmetic

17.1 Overview of IEEE arithmetic support

- 1 The intrinsic modules IEEE_EXCEPTIONS, IEEE_ARITHMETIC, and IEEE_FEATURES provide support for the facilities defined by ISO/IEC 60559:2020*. Whether the modules are provided is processor dependent. If the module IEEE_FEATURES is provided, which of the [named constants](#) defined in this document are included is processor dependent. The module IEEE_ARITHMETIC behaves as if it contained a [USE statement](#) for IEEE_EXCEPTIONS; everything that is public in IEEE_EXCEPTIONS is public in IEEE_ARITHMETIC.

NOTE 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 in the [scoping unit](#) can be determined by the function IEEE_SUPPORT_FLAG ([17.11.55](#)); whether control of [halting](#) is supported can be determined by the function IEEE_SUPPORT_HALTING. The extent of support of the other exceptions can 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 IEEE_UNDERFLOW_FLAG is accessible, within the [scoping unit](#) the processor shall support underflow 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 for at least one kind of real. If IEEE_HALTING is accessible, within the [scoping unit](#) the processor shall support control of [halting](#).

NOTE 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 60559:2020 to provide support for overflow and divide_by_zero. On a processor which does support ISO/IEC 60559:2020, 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 60559:2020 facilities are available from the module IEEE_ARITHMETIC. The extent of support can be influenced by the accessibility of the [named constants](#) of the module IEEE_FEATURES. If IEEE_DATATYPE of IEEE_FEATURES is accessible, within the [scoping unit](#) the processor shall support IEEE arithmetic 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 for at least one kind of real. In the case of IEEE_ROUNDING, it shall support 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_DATATYPE](#) (X) is true

* Because ISO/IEC 60559:2020 was originally an IEEE standard, its facilities are widely known as “IEEE arithmetic”, and this terminology is used by this document.

and **RADIX** (X) is equal to ten. Note that the effect of IEEE_DENORMAL is the same as that of IEEE_SUBNORMAL.

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.

17.2 Derived types, constants, and operators defined in the modules

The modules IEEE_EXCEPTIONS, IEEE_ARITHMETIC, and IEEE_FEATURES define derived types whose components are all private. No **direct component** of any of these types is **allocatable** or a pointer.

The module IEEE_EXCEPTIONS defines the following types and constants.

- 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.

The module IEEE_ARITHMETIC defines the following types, constants, and operators.

- 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_SUBNORMAL, 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, **IEEE_AWAY** and IEEE_OTHER for the rounding modes specified in this document.
- The **simple elemental operator** == for two values of one of these types to return true if the values are the same and false otherwise.
- The **simple elemental operator** /= for two values of one of these types to return true if the values differ and false otherwise.

The module IEEE_FEATURES defines the following types and constants.

- The type IEEE_FEATURES_TYPE, for expressing the need for particular ISO/IEC/IEEE 60559:2020 features. Its only possible values are those of **named constants** defined in the module: IEEE_DATA_TYPE, 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.

17.3 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 60559:2020 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 60559:2020 if `IEEE_SUPPORT_DATATYPE` is true for the operands of the division, and as determined by the processor otherwise. It is processor-dependent whether it occurs in a real exponentiation with a negative exponent. It occurs in a complex division if it is caused by the calculation of the real or imaginary part of the result.
- IEEE_INVALID occurs when a real or complex operation or assignment is invalid; possible examples are `SQRT (X)` when `X` is real and has a nonzero negative value, and conversion to an integer (by assignment, an intrinsic procedure, or a procedure defined in an intrinsic module) when the result is too large to be representable. IEEE_INVALID occurs for numeric relational intrinsic operations as specified below.
- IEEE_UNDERFLOW occurs when the result for an intrinsic real operation or assignment has an absolute value less than a processor-dependent limit, 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.
- 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. It is set to signaling when the associated exception occurs, except that the flag for IEEE_UNDERFLOW is not set if the result of the operation that caused the exception was exact and default ISO/IEC/IEEE 60559:2020 exception handling is in effect for IEEE_UNDERFLOW. 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. If a flag signals during execution of a procedure, the processor shall not set it to quiet on return.

4 Evaluation of a `specification expression` might cause an exception to signal.

5 In a `scoping unit` that has access to `IEEE_EXCEPTIONS` or `IEEE_ARITHMETIC`, if an intrinsic procedure or a procedure defined in an intrinsic module executes normally, the values of the flags `IEEE_OVERFLOW`, `IEEE_DIVIDE_BY_ZERO`, and `IEEE_INVALID` shall be as on entry to the procedure, even if one or more of them signals during the calculation. If a real or complex result is too large for the procedure to handle, `IEEE_OVERFLOW` may signal. If a real or complex result is a `NaN` because of an invalid operation (for example, `LOG (−1.0)`), `IEEE_INVALID` may signal. Similar rules apply to format processing and to intrinsic operations: no signaling flag shall be set quiet and no quiet flag shall be set signaling because of an intermediate calculation that does not affect the result.

6 In a scoping unit that has access to `IEEE_EXCEPTIONS` or `IEEE_ARITHMETIC`, if x_1 and x_2 are numeric entities

ies, the type of $x_1 + x_2$ is real, and `IEEE_SUPPORT_NAN` ($x_1 + x_2$) is true, the relational intrinsic operation x_1 *rel-op* x_2 shall signal `IEEE_INVALID` as specified for the conditional predicate of ISO/IEC 60559:2020 corresponding to *rel-op* indicated by Table 17.1. If the types or kind type parameters of x_1 or x_2 differ, the conversions (10.1.5.5.1) might signal exceptions instead of or in addition to an `IEEE_INVALID` exception signaled by the comparison.

NOTE

Each comparison predicate defined by ISO/IEC 60559:2020 is either unordered signaling or unordered quiet. An unordered signaling predicate signals an invalid operation exception if and only if one of the values being compared is a NaN. An unordered quiet predicate signals an invalid operation exception if and only if one of the values being compared is a signaling NaN. The comparison predicates do not signal any other exceptions.

Table 17.1 — IEEE relational operator correspondence

Operator	ISO/IEC/IEEE 60559:2020 comparison predicate
<code>.LT.</code> or <code><</code>	<code>compareSignalingLess</code>
<code>.LE.</code> or <code><=</code>	<code>compareSignalingLessEqual</code>
<code>.GT.</code> or <code>></code>	<code>compareSignalingGreater</code>
<code>.GE.</code> or <code>>=</code>	<code>compareSignalingGreaterEqual</code>
<code>.EQ.</code> or <code>==</code>	<code>compareQuietEqual</code>
<code>.NE.</code> or <code>/=</code>	<code>compareQuietNotEqual</code>

In a scoping unit that has access to `IEEE_EXCEPTIONS` or `IEEE_ARITHMETIC`, if x_1 or x_2 are numeric entities, the type of $x_1 + x_2$ is complex, and `IEEE_SUPPORT_NAN` (`REAL` ($x_1 + x_2$)) is true, the intrinsic equality or inequality operation between x_1 and x_2 may signal `IEEE_INVALID` if the value of the real or imaginary part of either operand is a signaling NaN. If any conversions are done before the values are compared, those conversions might signal exceptions instead of or in addition to an `IEEE_INVALID` exception signaled by the comparison.

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.

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
```


1 shall not signal IEEE_DIVIDE_BY_ZERO. On the other hand, when X has the value 1.0 and Y has the value
2 0.0, the expression

3 $X > 0.00001$.OR. $X/Y > 0.00001$

4 is permitted to cause the signaling of IEEE_DIVIDE_BY_ZERO.

5 10 The processor need not support IEEE_INVALID, IEEE_UNDERFLOW, and IEEE_INEXACT. If an exception is
6 not supported, its flag is always quiet.

7 17.4 The rounding modes

8 1 This document specifies a binary rounding mode that affects floating-point arithmetic with radix two, and
9 a decimal rounding mode that affects floating-point arithmetic with radix ten. Unqualified references to
10 the rounding mode with respect to a particular arithmetic operation or operands refers to the mode for the
11 radix of the operation or operands, and other unqualified references to the rounding mode refers to both
12 binary and decimal rounding modes.

13 2 ISO/IEC 60559:2020 specifies five possible rounding-direction attributes: roundTiesToEven, roundTo-
14 wardZero, roundTowardPositive, roundTowardNegative, and roundTiesToAway. These correspond to the
15 rounding modes IEEE_NEAREST, IEEE_TO_ZERO, IEEE_UP, IEEE_DOWN, and IEEE_AWAY respectively. The
16 rounding mode IEEE_OTHER does not correspond to any ISO/IEC/IEEE 60559:2020 rounding-direction
17 attribute; if supported, the effect of this rounding mode is processor dependent.

18 3 The subroutine [IEEE_GET_ROUNDING_MODE](#) can be used to get the rounding modes. The initial rounding
19 modes are processor dependent.

20 4 If the processor supports the alteration of the rounding modes during execution, the subroutine [IEEE_-](#)
21 [SET_ROUNDING_MODE](#) can be used to alter them.

22 5 In a procedure other than [IEEE_SET_ROUNDING_MODE](#) or [IEEE_SET_STATUS](#), the processor shall not
23 change the rounding modes on entry, and on return shall ensure that the rounding modes are the same
24 as they were on entry.

NOTE 1

ISO/IEC 60559:2020 requires support for roundTiesToAway only for decimal floating-point.

NOTE 2

ISO/IEC 60559:2020 requires that there is a language-defined means to specify a constant value for the rounding-direction attribute for all standard operations in a block. The means provided by this document are a [CALL](#) to [IEEE_GET_ROUNDING_MODE](#) at the beginning of the block followed by a [CALL](#) to [IEEE_SET_ROUNDING_MODE](#) with constant arguments, together with another [CALL](#) to [IEEE_SET_ROUNDING_MODE](#) at the end of the block to restore the rounding mode.

NOTE 3

Within a program, all literal constants that have the same form have the same value (7.1.4). Therefore, the value of a literal constant is not affected by the rounding modes.

17.5 Underflow mode

- 1 Some processors allow control during program execution of whether underflow produces a subnormal number in conformance with ISO/IEC 60559:2020 (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.

17.6 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 initial halting mode is processor dependent. In a procedure other than IEEE_SET_HALTING_MODE or IEEE_SET_STATUS, the processor shall not change the halting mode on entry, and on return shall ensure that the halting mode is the same as it was on entry.

17.7 The floating-point modes and status

- 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 1

Each [image](#) has its own floating-point status (5.3.4).

NOTE 2

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 3

The processor is required to ensure that a call to a Fortran procedure does not change the floating-point status other than by setting exception flags to signaling.

17.8 Exceptional values

ISO/IEC 60559:2020 specifies the following exceptional floating-point values.

- Subnormal values have very small absolute values and reduced precision.
- Infinite values (+infinity and –infinity) are created by overflow or division by zero.
- Not-a-Number (NaN) values are undefined values or values created by an invalid operation.

A value that does not fall into the above classes is called a normal number.

The functions IEEE_IS_FINITE, IEEE_IS_NAN, IEEE_IS_NEGATIVE, and IEEE_IS_NORMAL are provided to test whether a value is finite, NaN, negative, or normal. The function IEEE_VALUE is provided to generate an IEEE number of any class, including an infinity or a NaN. The inquiry functions IEEE_SUPPORT_SUBNORMAL, IEEE_SUPPORT_INF, and IEEE_SUPPORT_NAN are provided to determine whether these facilities are available for a particular kind of real.

17.9 IEEE arithmetic

The inquiry function IEEE_SUPPORT_DATATYPE can be used to inquire whether IEEE arithmetic is supported for a particular kind of real. Complete conformance with ISO/IEC 60559:2020 is not required, but

- the normal numbers shall be exactly those of an ISO/IEC/IEEE 60559:2020 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 60559:2020 are normal numbers,
- the IEEE function abs shall be provided by the intrinsic function ABS,
- the IEEE operation remainder shall be provided by the function IEEE_REM, and
- the IEEE functions copySign, logB, scaleB, and compareQuietUnordered shall be provided by the functions IEEE_COPY_SIGN, IEEE_LOGB, IEEE_SCALB, and IEEE_UNORDERED, respectively,

for that kind of real.

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 60559:2020 when the result is an IEEE NaN.

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 60559:2020 when exactly one operand or the result specified by ISO/IEC 60559:2020 is an IEEE infinity.

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 60559:2020 when the result specified by ISO/IEC 60559:2020 is subnormal, or any operand is subnormal and either the result is not an IEEE infinity or IEEE_SUPPORT_INF is true.

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 60559:2020

when both operands and the result specified by ISO/IEC 60559:2020 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 60559:2020 when the result specified by ISO/IEC 60559:2020 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 60559:2020 when one operand or the result specified by ISO/IEC 60559:2020 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 60559:2020 when the result specified by ISO/IEC 60559:2020 is subnormal, or when any operand is subnormal and either the result specified by ISO/IEC 60559:2020 is not an infinity or IEEE_SUPPORT_INF is true.

ISO/IEC 60559:2020 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 60559:2020 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 60559:2020 when the result specified by ISO/IEC 60559:2020 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 60559:2020 when the result specified by ISO/IEC 60559:2020 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 60559:2020 when the argument is subnormal.

The inquiry function IEEE_SUPPORT_STANDARD is provided to inquire whether the processor supports all the ISO/IEC/IEEE 60559:2020 facilities defined in this document for a particular kind of real.

17.10 Summary of the procedures

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.

A procedure classified in 17.10 as an inquiry function depends on the properties of one or more of its arguments instead of their values; in fact, these argument values may be undefined. Unless the description of one of these inquiry functions states otherwise, these arguments are permitted to be unallocated allocatable variables or pointers that are undefined or disassociated. A procedure that is classified as a transformational function is neither an inquiry function nor elemental.

In the Class column of Tables 17.2 and 17.3,

E indicates that the procedure is an elemental function,

ES indicates that the procedure is a simple elemental subroutine,

I indicates that the procedure is an inquiry function,

SS indicates that the procedure is a simple subroutine, and

T indicates that the procedure is a transformational function.

Table 17.2 — 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])	SS	Get rounding mode.
IEEE_GET_UNDERFLOW_MODE (GRADUAL)	SS	Get underflow mode.
IEEE_INT (A, ROUND [, KIND])	E	Conversion to integer type.

Table 17.2: IEEE_ARITHMETIC module procedure summary (cont.)

Procedure (arguments)	Class	Description
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 (X, Y)	E	Maximum value.
IEEE_MAX_MAG (X, Y)	E	Maximum magnitude value.
IEEE_MAX_NUM (X, Y)	E	Maximum numeric value.
IEEE_MAX_NUM_MAG (X, Y)	E	Maximum magnitude numeric value.
IEEE_MIN (X, Y)	E	Minimum value.
IEEE_MIN_MAG (X, Y)	E	Minimum magnitude 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.
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])	SS	Set rounding mode.
IEEE_SET_UNDERFLOW_MODE (GRADUAL)	SS	Set underflow mode .
IEEE_SIGNALING_EQ (A, B)	E	Signaling compares equal.
IEEE_SIGNALING_GE (A, B)	E	Signaling compares greater than or equal.
IEEE_SIGNALING_GT (A, B)	E	Signaling compares greater than.
IEEE_SIGNALING_LE (A, B)	E	Signaling compares less than or equal.
IEEE_SIGNALING_LT (A, B)	E	Signaling compares less than.
IEEE_SIGNALING_NE (A, B)	E	Signaling compares not equal.
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])	T	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 17.3 — 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)	SS	Get floating-point modes.
IEEE_GET_STATUS (STATUS_VALUE)	SS	Get floating-point status.
IEEE_SET_FLAG (FLAG, FLAG_VALUE)	SS	Set an exception flag.
IEEE_SET_HALTING_MODE (FLAG, HALTING)	SS	Set a halting mode .
IEEE_SET_MODES (MODES)	SS	Set floating-point modes.
IEEE_SET_STATUS (STATUS_VALUE)	SS	Restore floating-point status.
IEEE_SUPPORT_FLAG (FLAG [, X])	T	Query exception support.
IEEE_SUPPORT_HALTING (FLAG)	T	Query halting mode support.

1 4 In the intrinsic module IEEE_ARITHMETIC, the [elemental](#) functions listed are provided for all reals X and
2 Y.

3 17.11 Specifications of the procedures

4 17.11.1 General

5 1 In the detailed descriptions in [17.11](#), procedure names are generic and are not specific. All the functions
6 are [simple](#) and all the subroutines are impure unless otherwise stated. All [dummy arguments](#) have [INTENT](#)
7 ([IN](#)) if the intent is not stated explicitly. In the examples, it is assumed that the processor supports IEEE
8 arithmetic for default real.

9 2 For the [elemental](#) functions of IEEE_ARITHMETIC that return a floating-point result, if X or Y has a value
10 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
11 60559:2020. For example, the result for an infinity shall be constructed as the limiting case of the result
12 with a value of arbitrarily large magnitude, if such a limit exists.

13 3 A program may contain statements that, if executed, would violate the requirements listed in a **Restriction**
14 paragraph.

NOTE

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.

15 17.11.2 IEEE_CLASS (X)

16 1 **Description.** Classify number.

17 2 **Class.** [Elemental](#) function.

18 3 **Argument.** X shall be of type real.

19 4 **Restriction.** IEEE_CLASS (X) shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the value false.

1 5 **Result Characteristics.** IEEE_CLASS_TYPE.

2 6 **Result Value.** The result value shall be IEEE_SIGNALING_NAN or IEEE_QUIET_NAN if IEEE_SUPPORT_-
 3 NAN (X) has the value true and the value of X is a signaling or quiet NaN, respectively. The result value
 4 shall be IEEE_NEGATIVE_INF or IEEE_POSITIVE_INF if IEEE_SUPPORT_INF (X) has the value true and the
 5 value of X is negative or positive infinity, respectively. The result value shall be IEEE_NEGATIVE_SUBNOR-
 6 MAL or IEEE_POSITIVE_SUBNORMAL if IEEE_SUPPORT_SUBNORMAL (X) has the value true and the value
 7 of X is a negative or positive subnormal value, respectively. The result value shall be IEEE_NEGATIVE_-
 8 NORMAL, IEEE_NEGATIVE_ZERO, IEEE_POSITIVE_ZERO, or IEEE_POSITIVE_NORMAL if the value of X is
 9 negative normal, negative zero, positive zero, or positive normal, respectively. Otherwise, the result value
 10 shall be IEEE_OTHER_VALUE.

11 7 **Example.** IEEE_CLASS (−1.0) has the value IEEE_NEGATIVE_NORMAL.

NOTE

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.

12 17.11.3 IEEE_COPY_SIGN (X, Y)

13 1 **Description.** Copy sign.

14 2 **Class.** Elemental function.

15 3 **Arguments.** The arguments shall be of type real.

16 4 **Restriction.** IEEE_COPY_SIGN (X, Y) shall not be invoked if IEEE_SUPPORT_DATATYPE (X) or IEEE_SUP-
 17 PORT_DATATYPE (Y) has the value false.

18 5 **Result Characteristics.** Same as X.

19 6 **Result Value.** The result has the absolute value of X with the sign of Y. This is true even for IEEE special
 20 values, such as a NaN or an infinity (on processors supporting such values).

21 7 **Example.** The value of IEEE_COPY_SIGN (X, 1.0) is ABS (X) even when X is a NaN.

22 17.11.4 IEEE_FMA (A, B, C)

23 1 **Description.** Fused multiply-add operation.

24 2 **Class.** Elemental function.

25 3 **Arguments.**

26 A shall be of type real.

27 B shall be of the same type and kind type parameter as A.

28 C shall be of the same type and kind type parameter as A.

29 4 **Restriction.** IEEE_FMA (A, B, C) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the value false.

30 5 **Result Characteristics.** Same as A.

31 6 **Result Value.** The result has the value specified by ISO/IEC 60559:2020 for the fusedMultiplyAdd opera-
 32 tion; 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.

Example. The value of IEEE_FMA (TINY (0.0), TINY (0.0), 1.0), when the rounding mode is IEEE_NEAREST, is equal to 1.0; only the IEEE_INEXACT exception is signaled.

17.11.5 IEEE_GET_FLAG (FLAG, FLAG_VALUE)

Description. Get an exception flag.

Class. Simple elemental subroutine.

Arguments.

FLAG shall be of type IEEE_FLAG_TYPE. It specifies the exception flag to be obtained.

FLAG_VALUE shall be of type logical. It is an INTENT (OUT) argument. If the value of FLAG is IEEE_INVALID, IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_UNDERFLOW, or IEEE_INEXACT, FLAG_VALUE is assigned the value true if the corresponding exception flag is signaling and is assigned the value false otherwise.

Example. Following CALL IEEE_GET_FLAG (IEEE_OVERFLOW, FLAG_VALUE), FLAG_VALUE is true if the IEEE_OVERFLOW flag is signaling and is false if it is quiet.

17.11.6 IEEE_GET_HALTING_MODE (FLAG, HALTING)

Description. Get a halting mode.

Class. Simple elemental subroutine.

Arguments.

FLAG shall be of type IEEE_FLAG_TYPE. It specifies the exception flag. It shall have one of the values IEEE_INVALID, IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_UNDERFLOW, or IEEE_INEXACT.

HALTING shall be of type logical. It is an INTENT (OUT) argument. It is assigned the value true if the exception specified by FLAG will cause halting. Otherwise, it is assigned the value false.

Example. To store the halting mode for IEEE_OVERFLOW, do a calculation without halting, and restore the halting mode later:

```
USE, INTRINSIC :: IEEE_ARITHMETIC
LOGICAL HALTING
...
CALL IEEE_GET_HALTING_MODE (IEEE_OVERFLOW, HALTING) ! Store halting mode
CALL IEEE_SET_HALTING_MODE (IEEE_OVERFLOW, .FALSE.) ! No halting
... ! calculation without halting
CALL IEEE_SET_HALTING_MODE (IEEE_OVERFLOW, HALTING) ! Restore halting mode
```

17.11.7 IEEE_GET_MODES (MODES)

Description. Get floating-point modes.

Class. Simple subroutine.

1 3 **Argument.** MODES shall be a scalar of type `IEEE_MODES_TYPE`. It is an **INTENT (OUT)** argument that is
2 assigned the value of the floating-point modes.

3 4 **Example.** To save the floating-point modes, do a calculation with specific rounding and **underflow** modes,
4 and restore them later:

```
5      USE, INTRINSIC :: IEEE_ARITHMETIC
6      TYPE (IEEE_MODES_TYPE) SAVE_MODES
7      ...
8      CALL IEEE_GET_MODES (SAVE_MODES) ! Save all modes.
9      CALL IEEE_SET_ROUNDING_MODE (IEEE_TO_ZERO)
10     CALL IEEE_SET_UNDERFLOW_MODE (GRADUAL=.FALSE.)
11     ... ! calculation with abrupt round-to-zero.
12     CALL IEEE_SET_MODES (SAVE_MODES) ! Restore all modes.
```

13 17.11.8 IEEE_GET_ROUNDING_MODE (ROUND_VALUE [, RADIX])

14 1 **Description.** Get rounding mode.

15 2 **Class.** **Simple** subroutine.

16 3 **Arguments.**

17 ROUND_VALUE shall be a scalar of type `IEEE_ROUND_TYPE`. It is an **INTENT (OUT)** argument. It is as-
18 signed the value `IEEE_NEAREST`, `IEEE_TO_ZERO`, `IEEE_UP`, `IEEE_DOWN`, or `IEEE_AWAY` if the
19 corresponding rounding mode is in operation and `IEEE_OTHER` otherwise.

20 RADIX (optional) shall be an integer scalar with the value two or ten. If RADIX is present with the value ten,
21 the rounding mode queried is the decimal rounding mode, otherwise it is the binary rounding
22 mode.

23 4 **Example.** To save the binary rounding mode, do a calculation with round to nearest, and restore the round-
24 ing mode later:

```
25      USE, INTRINSIC :: IEEE_ARITHMETIC
26      TYPE (IEEE_ROUND_TYPE) ROUND_VALUE
27      ...
28      CALL IEEE_GET_ROUNDING_MODE (ROUND_VALUE) ! Store the rounding mode
29      CALL IEEE_SET_ROUNDING_MODE (IEEE_NEAREST)
30      ... ! calculation with round to nearest
31      CALL IEEE_SET_ROUNDING_MODE (ROUND_VALUE) ! Restore the rounding mode
```

32 17.11.9 IEEE_GET_STATUS (STATUS_VALUE)

33 1 **Description.** Get floating-point status.

34 2 **Class.** **Simple** subroutine.

35 3 **Argument.** STATUS_VALUE shall be a scalar of type `IEEE_STATUS_TYPE`. It is an **INTENT (OUT)** argument.
36 It is assigned the value of the floating-point status.

37 4 **Example.** To store all the exception flags, do a calculation involving exception handling, and restore them
38 later:

```
39      USE, INTRINSIC :: IEEE_ARITHMETIC
40      TYPE (IEEE_STATUS_TYPE) STATUS_VALUE
```

```

1      ...
2      CALL IEEE_GET_STATUS (STATUS_VALUE) ! Get the flags
3      CALL IEEE_SET_FLAG (IEEE_ALL, .FALSE.) ! Set the flags quiet.
4      ... ! calculation involving exception handling
5      CALL IEEE_SET_STATUS (STATUS_VALUE) ! Restore the flags

```

17.11.10 IEEE_GET_UNDERFLOW_MODE (GRADUAL)

1 **Description.** Get [underflow mode](#).

2 **Class.** [Simple](#) subroutine.

3 **Argument.** GRADUAL shall be a logical scalar. It is an [INTENT \(OUT\)](#) argument. It is assigned the value true if the [underflow mode](#) is gradual underflow, and false if the [underflow mode](#) is abrupt underflow.

4 **Restriction.** IEEE_GET_UNDERFLOW_MODE shall not be invoked unless IEEE_SUPPORT_UNDERFLOW_CONTROL (X) is true for some X.

5 **Example.** After CALL [IEEE_SET_UNDERFLOW_MODE](#) (.FALSE.), a subsequent CALL IEEE_GET_UNDERFLOW_MODE (GRADUAL) will set GRADUAL to false.

17.11.11 IEEE_INT (A, ROUND [, KIND])

1 **Description.** Conversion to integer type.

2 **Class.** [Elemental](#) function.

3 **Arguments.**

A shall be of type real.

ROUND shall be of type [IEEE_ROUND_TYPE](#).

KIND (optional) shall be a scalar integer constant expression.

4 **Restriction.** IEEE_INT (A, ROUND, KIND) shall not be invoked if [IEEE_SUPPORT_DATATYPE](#) (A) has the value false.

5 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default integer.

6 **Result Value.** The result has the value specified by ISO/IEC 60559:2020 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.

17.11.12 IEEE_IS_FINITE (X)

Description. Whether a value is finite.

Class. [Elemental](#) function.

Argument. X shall be of type real.

Restriction. IEEE_IS_FINITE (X) shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the value false.

Result Characteristics. Default logical.

Result Value. The result has the value true if the value of X is finite, that is, IEEE_CLASS (X) has one of the values IEEE_NEGATIVE_NORMAL, [IEEE_NEGATIVE_SUBNORMAL](#), IEEE_NEGATIVE_ZERO, IEEE_POSITIVE_ZERO, [IEEE_POSITIVE_SUBNORMAL](#), or IEEE_POSITIVE_NORMAL; otherwise, the result has the value false.

Example. IEEE_IS_FINITE (1.0) has the value true.

17.11.13 IEEE_IS_NAN (X)

Description. Whether a value is an [IEEE NaN](#).

Class. [Elemental](#) function.

Argument. X shall be of type real.

Restriction. IEEE_IS_NAN (X) shall not be invoked if IEEE_SUPPORT_NAN (X) has the value false.

Result Characteristics. Default logical.

Result Value. The result has the value true if the value of X is an [IEEE NaN](#); otherwise, it has the value false.

Example. IEEE_IS_NAN ([SQRT](#) (−1.0)) has the value true if IEEE_SUPPORT_SQRT (1.0) has the value true.

17.11.14 IEEE_IS_NEGATIVE (X)

Description. Whether a value is negative.

Class. [Elemental](#) function.

Argument. X shall be of type real.

Restriction. IEEE_IS_NEGATIVE (X) shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the value false.

Result Characteristics. Default logical.

Result Value. The result has the value true if IEEE_CLASS (X) has one of the values IEEE_NEGATIVE_NORMAL, [IEEE_NEGATIVE_SUBNORMAL](#), IEEE_NEGATIVE_ZERO or IEEE_NEGATIVE_INF; otherwise, the result has the value false.

Example. IEEE_IS_NEGATIVE (0.0) has the value false.

17.11.15 IEEE_IS_NORMAL (X)

Description. Whether a value is a normal number.

Class. [Elemental](#) function.

Argument. X shall be of type real.

Restriction. IEEE_IS_NORMAL (X) shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the value false.

Result Characteristics. Default logical.

Result Value. The result has the value true if IEEE_CLASS (X) has one of the values IEEE_NEGATIVE_NORMAL, IEEE_NEGATIVE_ZERO, IEEE_POSITIVE_ZERO or IEEE_POSITIVE_NORMAL; otherwise, the result has the value false.

Example. IEEE_IS_NORMAL ([SQRT](#) (−1.0)) has the value false if IEEE_SUPPORT_SQRT (1.0) has the value true.

17.11.16 IEEE_LOGB (X)

Description. Exponent.

Class. [Elemental](#) function.

Argument. X shall be of type real.

Restriction. IEEE_LOGB (X) shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the value false.

Result Characteristics. Same as X.

Result Value.

Case (i): If the value of X is neither zero, infinity, nor NaN, the result has the value of the unbiased exponent of X. Note: this value is equal to [EXPONENT](#) (X) − 1.

Case (ii): If X==0, the result is −infinity if IEEE_SUPPORT_INF (X) is true and −[HUGE](#) (X) otherwise; IEEE_DIVIDE_BY_ZERO signals.

Case (iii): If IEEE_SUPPORT_INF (X) is true and X is infinite, the result is +infinity.

Case (iv): If IEEE_SUPPORT_NAN (X) is true and X is a NaN, the result is a NaN.

Example. IEEE_LOGB (−1.1) has the value 0.0.

17.11.17 IEEE_MAX (X, Y)

Description. Maximum value.

Class. [Elemental](#) function.

Arguments.

X shall be of type real.

Y shall be of the same type and kind type parameter as X.

Restriction. IEEE_MAX shall not be invoked if [IEEE_SUPPORT_DATATYPE](#) (X) has the value false.

Result Characteristics. Same as X.

Result Value. The result has the value specified for the maximum operation in ISO/IEC 60559:2020; that is,

- if $X > Y$ the result has the value of X ;
- if $Y > X$ the result has the value of Y ;
- if either operand is a NaN, the result is a quiet NaN;
- if $X = Y$ and the signs are the same, the result is the value of either X or Y ;
- otherwise (one argument is negative zero and the other is positive zero), the result is positive zero.

If one or both of X and Y are signaling NaNs, IEEE_INVALID signals; otherwise, no exception is signaled.

Example. The value of IEEE_MAX (1.5, IEEE_VALUE (1.0, IEEE_QUIET_NAN)) is a quiet NaN.

17.11.18 IEEE_MAX_MAG (X, Y)

Description. Maximum magnitude value.

Class. Elemental function.

Arguments.

X shall be of type real.

Y shall be of the same type and kind type parameter as X .

Restriction. IEEE_MAX_MAG shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the value false.

Result Characteristics. Same as X .

Result Value.

The result has the value specified for the maximumMagnitude operation in ISO/IEC 60559:2020; that is,

- if $|X| > |Y|$ the result has the value of X ;
- if $|Y| > |X|$ the result has the value of Y ;
- otherwise, the result has the value of IEEE_MAX (X , Y).

If one or both of X and Y are signaling NaNs, IEEE_INVALID signals; otherwise, no exception is signaled.

Example. The value of IEEE_MAX_MAG (1.5, -2.5) is -2.5 .

17.11.19 IEEE_MAX_NUM (X, Y)

Description. Maximum numeric value.

Class. Elemental function.

Arguments.

X shall be of type real.

Y shall be of the same type and kind type parameter as X .

Restriction. IEEE_MAX_NUM shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the value false.

Result Characteristics. Same as X .

Result Value. The result has the value specified for the maximumNumber operation in ISO/IEC 60559:2020; 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 NaN the result has the value of the other argument;
- if both X and Y are NaNs, the result is a quiet NaN;
- if $X = Y$ and the signs are the same, the result is either X or Y ;
- otherwise (one argument is negative zero and the other is positive zero), the result is positive zero.

If one or both of X and Y are signaling NaNs, IEEE_INVALID signals, but unless X and Y are both signaling NaNs, the signaling NaN is otherwise ignored and not converted to a quiet NaN. No other exceptions are signaled.

Example. The value of IEEE_MAX_NUM (1.5, IEEE_VALUE (1.0, IEEE_QUIET_NAN)) is 1.5.

17.11.20 IEEE_MAX_NUM_MAG (X, Y)

Description. Maximum magnitude numeric value.

Class. Elemental function.

Arguments.

X shall be of type real.

Y shall be of the same type and kind type parameter as X .

Restriction. IEEE_MAX_NUM_MAG shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the value false.

Result Characteristics. Same as X .

Result Value. The result has the value specified for the maximumMagnitudeNumber operation in ISO/IEC 60559:2020; that is,

- if $|X| > |Y|$ the result has the value of X ;
- if $|Y| > |X|$ the result has the value of Y ;
- otherwise, the result has the value of IEEE_MAX_NUM (X, Y).

If one or both of X and Y are signaling NaNs, IEEE_INVALID signals, but unless X and Y are both signaling NaNs, the signaling NaN is otherwise ignored and not converted to a quiet NaN. No other exceptions are signaled.

Example. The value of IEEE_MAX_NUM_MAG (1.5, -2.5) is -2.5 .

17.11.21 IEEE_MIN (X, Y)

Description. Minimum value.

Class. Elemental function.

Arguments.

X shall be of type real.

Y shall be of the same type and kind type parameter as X .

Restriction. IEEE_MIN shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the value false.

1 5 **Result Characteristics.** Same as X.

2 6 **Result Value.** The result has the value specified for the minimum operation in ISO/IEC 60559:2020; that
3 is,

- 4 • if $X < Y$ the result has the value of X;
- 5 • if $Y < X$ the result has the value of Y;
- 6 • if either operand is a NaN, the result is a quiet NaN;
- 7 • if $X = Y$ and the signs are the same, the result is the value of either X or Y;
- 8 • otherwise (one argument is negative zero and the other is positive zero), the result is negative zero.

9 If one or both of X and Y are signaling NaNs, IEEE_INVALID signals; otherwise, no exception is signaled.

10 7 **Example.** The value of IEEE_MIN (1.5, IEEE_VALUE (1.0, IEEE_QUIET_NAN)) is a quiet NaN.

11 17.11.22 IEEE_MIN_MAG (X, Y)

12 1 **Description.** Minimum magnitude value.

13 2 **Class.** Elemental function.

14 3 **Arguments.**

15 X shall be of type real.

16 Y shall be of the same type and kind type parameter as X.

17 4 **Restriction.** IEEE_MIN_MAG shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the value false.

18 5 **Result Characteristics.** Same as X.

19 6 **Result Value.**

20 7 The result has the value specified for the minimumMagnitude operation in ISO/IEC 60559:2020; that is,

- 21 • if $|X| < |Y|$ the result has the value of X;
- 22 • if $|Y| < |X|$ the result has the value of Y;
- 23 • otherwise, the result has the value of IEEE_MIN (X, Y).

24 If one or both of X and Y are signaling NaNs, IEEE_INVALID signals; otherwise, no exception is signaled.

25 8 **Example.** The value of IEEE_MIN_MAG (1.5, −2.5) is 1.5.

26 17.11.23 IEEE_MIN_NUM (X, Y)

27 1 **Description.** Minimum numeric value.

28 2 **Class.** Elemental function.

29 3 **Arguments.**

30 X shall be of type real.

31 Y shall be of the same type and kind type parameter as X.

32 4 **Restriction.** IEEE_MIN_NUM shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the value false.

33 5 **Result Characteristics.** Same as X.

Result Value. The result has the value specified for the minimumNumber operation in ISO/IEC 60559:2020; 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 NaN the result has the value of the other argument;
- if both X and Y are NaNs, the result is a quiet NaN;
- if $X = Y$ and the signs are the same, the result is either X or Y ;
- otherwise (one argument is negative zero and the other is positive zero), the result is negative zero.

If one or both of X and Y are signaling NaNs, IEEE_INVALID signals, but unless X and Y are both signaling NaNs, the signaling NaN is otherwise ignored and not converted to a quiet NaN. No other exceptions are signaled.

Example. The value of IEEE_MIN_NUM (1.5, IEEE_VALUE (1.0, IEEE_QUIET_NAN)) is 1.5.

17.11.24 IEEE_MIN_NUM_MAG (X, Y)

Description. Minimum magnitude numeric value.

Class. Elemental function.

Arguments.

X shall be of type real.

Y shall be of the same type and kind type parameter as X .

Restriction. IEEE_MIN_NUM_MAG shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the value false.

Result Characteristics. Same as X .

Result Value. The result has the value specified for the minimumMagnitudeNumber operation in ISO/IEC 60559:2020; that is,

- if $|X| < |Y|$ the result has the value of X ;
- if $|Y| < |X|$ the result has the value of Y ;
- otherwise, the result has the value of IEEE_MIN_NUM (X, Y).

If one or both of X and Y are signaling NaNs, IEEE_INVALID signals, but unless X and Y are both signaling NaNs, the signaling NaN is otherwise ignored and not converted to a quiet NaN. No other exceptions are signaled.

Example. The value of IEEE_MIN_NUM_MAG (1.5, -2.5) is 1.5.

17.11.25 IEEE_NEXT_AFTER (X, Y)

Description. Adjacent machine number.

Class. Elemental function.

Arguments. The arguments shall be of type real.

Restriction. IEEE_NEXT_AFTER (X, Y) shall not be invoked if IEEE_SUPPORT_DATATYPE (X) or IEEE_SUPPORT_DATATYPE (Y) has the value false.

1 5 **Result Characteristics.** Same as X.

2 6 **Result Value.**

3 *Case (i):* If $X == Y$, the result is X and no exception is signaled.

4 *Case (ii):* If $X \neq Y$, the result has the value of the next representable neighbor of X in the direction of Y.
 5 The neighbors of zero (of either sign) are both nonzero. IEEE_OVERFLOW is signaled when X
 6 is finite but IEEE_NEXT_AFTER (X, Y) is infinite; IEEE_UNDERFLOW is signaled when IEEE_
 7 NEXT_AFTER (X, Y) is subnormal; in both cases, IEEE_INEXACT signals.

8 7 **Example.** The value of IEEE_NEXT_AFTER (1.0, 2.0) is $1.0 + \text{EPSILON}(X)$.

9 17.11.26 IEEE_NEXT_DOWN (X)

10 1 **Description.** Adjacent lower machine number.

11 2 **Class.** Elemental function.

12 3 **Argument.** X shall be of type real.

13 4 **Restriction.** IEEE_NEXT_DOWN (X) shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the value
 14 false. IEEE_NEXT_DOWN ($-\text{HUGE}(X)$) shall not be invoked if IEEE_SUPPORT_INF (X) has the value false.

15 5 **Result Characteristics.** Same as X.

16 6 **Result Value.** The result has the value specified for the nextDown operation in ISO/IEC 60559:2020; that
 17 is, it is the greatest value in the representation method of X that compares less than X, except when X is
 18 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,
 19 IEEE_INVALID signals; otherwise, no exception is signaled.

20 7 **Example.** If IEEE_SUPPORT_SUBNORMAL (0.0) is true, the value of IEEE_NEXT_DOWN (+0.0) is the neg-
 21 ative subnormal number with least magnitude.

22 17.11.27 IEEE_NEXT_UP (X)

23 1 **Description.** Adjacent higher machine number.

24 2 **Class.** Elemental function.

25 3 **Argument.** X shall be of type real.

26 4 **Restriction.** IEEE_NEXT_UP (X) shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the value false.
 27 IEEE_NEXT_UP ($\text{HUGE}(X)$) shall not be invoked if IEEE_SUPPORT_INF (X) has the value false.

28 5 **Result Characteristics.** Same as X.

29 6 **Result Value.** The result has the value specified for the nextUp operation in ISO/IEC 60559:2020; that
 30 is, it is the least value in the representation method of X that compares greater than X, except when X is
 31 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,
 32 IEEE_INVALID signals; otherwise, no exception is signaled.

33 7 **Example.** If IEEE_SUPPORT_INF (X) is true, the value of IEEE_NEXT_UP ($\text{HUGE}(X)$) is $+\infty$.

17.11.28 IEEE_QUIET_EQ (A, B)

Description. Quiet compares equal.

Class. [Elemental](#) function.

Arguments.

A shall be of type real.

B shall have the same type and kind type parameter as A.

Restriction. IEEE_QUIET_EQ (A, B) shall not be invoked if [IEEE_SUPPORT_DATATYPE](#) (A) has the value false.

Result Characteristics. Default logical.

Result Value. The result has the value specified for the compareQuietEqual operation in ISO/IEC 60559:2020; that is, it is true if and only if A compares equal to B. If A or B is a NaN, the result will be false. If A or B is a signaling NaN, IEEE_INVALID signals; otherwise, no exception is signaled.

Example. IEEE_QUIET_EQ (1.0, [IEEE_VALUE](#) (1.0, IEEE_QUIET_NAN)) has the value false and no exception is signaled.

17.11.29 IEEE_QUIET_GE (A, B)

Description. Quiet compares greater than or equal.

Class. [Elemental](#) function.

Arguments.

A shall be of type real.

B shall have the same type and kind type parameter as A.

Restriction. IEEE_QUIET_GE (A, B) shall not be invoked if [IEEE_SUPPORT_DATATYPE](#) (A) has the value false.

Result Characteristics. Default logical.

Result Value. The result has the value specified for the compareQuietGreaterEqual operation in ISO/IEC 60559:2020; that is, it is true if and only if A compares greater than or equal to B. If A or B is a NaN, the result will be false. If A or B is a signaling NaN, IEEE_INVALID signals; otherwise, no exception is signaled.

Example. IEEE_QUIET_GE (1.0, [IEEE_VALUE](#) (1.0, IEEE_QUIET_NAN)) has the value false and no exception is signaled.

17.11.30 IEEE_QUIET_GT (A, B)

Description. Quiet compares greater than.

Class. [Elemental](#) function.

Arguments.

A shall be of type real.

B shall have the same type and kind type parameter as A.

1 4 **Restriction.** IEEE_QUIET_GT (A, B) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the value
2 false.

3 5 **Result Characteristics.** Default logical.

4 6 **Result Value.** The result has the value specified for the compareQuietGreater operation in ISO/IEC
5 60559:2020; that is, it is true if and only if A compares greater than B. If A or B is a NaN, the result will
6 be false. If A or B is a signaling NaN, IEEE_INVALID signals; otherwise, no exception is signaled.

7 7 **Example.** IEEE_QUIET_GT (1.0, IEEE_VALUE (1.0, IEEE_QUIET_NAN)) has the value false and no exception
8 is signaled.

9 17.11.31 IEEE_QUIET_LE (A, B)

10 1 **Description.** Quiet compares less than or equal.

11 2 **Class.** Elemental function.

12 3 **Arguments.**

13 A shall be of type real.

14 B shall have the same type and kind type parameter as A.

15 4 **Restriction.** IEEE_QUIET_LE (A, B) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the value
16 false.

17 5 **Result Characteristics.** Default logical.

18 6 **Result Value.** The result has the value specified for the compareQuietLessEqual operation in ISO/IEC
19 60559:2020; 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
20 result will be false. If A or B is a signaling NaN, IEEE_INVALID signals; otherwise, no exception is signaled.

21 7 **Example.** IEEE_QUIET_LE (1.0, IEEE_VALUE (1.0, IEEE_QUIET_NAN)) has the value false and no exception
22 is signaled.

23 17.11.32 IEEE_QUIET_LT (A, B)

24 1 **Description.** Quiet compares less than.

25 2 **Class.** Elemental function.

26 3 **Arguments.**

27 A shall be of type real.

28 B shall have the same type and kind type parameter as A.

29 4 **Restriction.** IEEE_QUIET_LT (A, B) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the value
30 false.

31 5 **Result Characteristics.** Default logical.

32 6 **Result Value.** The result has the value specified for the compareQuietLess operation in ISO/IEC
33 60559:2020; 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
34 false. If A or B is a signaling NaN, IEEE_INVALID signals; otherwise, no exception is signaled.

1 7 **Example.** IEEE_QUIET_LT (1.0, IEEE_VALUE (1.0, IEEE_QUIET_NAN)) has the value false and no exception
2 is signaled.

3 17.11.33 IEEE_QUIET_NE (A, B)

4 1 **Description.** Quiet compares not equal.

5 2 **Class.** Elemental function.

6 3 **Arguments.**

7 A shall be of type real.

8 B shall have the same type and kind type parameter as A.

9 4 **Restriction.** IEEE_QUIET_NE (A, B) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the value
10 false.

11 5 **Result Characteristics.** Default logical.

12 6 **Result Value.** The result has the value specified for the compareQuietNotEqual operation in ISO/IEC
13 60559:2020; 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
14 be true. If A or B is a signaling NaN, IEEE_INVALID signals; otherwise, no exception is signaled.

15 7 **Example.** IEEE_QUIET_NE (1.0, IEEE_VALUE (1.0, IEEE_QUIET_NAN)) has the value true and no exception
16 is signaled.

17 17.11.34 IEEE_REAL (A [, KIND])

18 1 **Description.** Conversion to real type.

19 2 **Class.** Elemental function.

20 3 **Arguments.**

21 A shall be of type integer or real.

22 KIND (optional) shall be a scalar integer constant expression.

23 4 **Restriction.** IEEE_REAL shall not be invoked if A is of type real and IEEE_SUPPORT_DATATYPE (A) has the
24 value false, or if IEEE_SUPPORT_DATATYPE (IEEE_REAL (A, KIND)) has the value false.

25 5 **Result Characteristics.** Real. If KIND is present, the kind type parameter is that specified by the value of
26 KIND; otherwise, the kind type parameter is that of default real.

27 6 **Result Value.** The result has the same value as A if that value is representable in the representation method
28 of the result, and is rounded according to the rounding mode otherwise. This shall be consistent with the
29 specification of ISO/IEC 60559:2020 for the convertFromInt operation when A is of type integer, and with
30 the convertFormat operation otherwise.

31 7 **Example.** The value of IEEE_REAL (123) is 123.0.

32 17.11.35 IEEE_REM (X, Y)

33 1 **Description.** Exact remainder.

34 2 **Class.** Elemental function.

- 1 3 **Arguments.** The arguments shall be of type real and have the same radix.
- 2 4 **Restriction.** IEEE_REM (X, Y) shall not be invoked if IEEE_SUPPORT_DATATYPE (X) or IEEE_SUPPORT_ -
3 DATATYPE (Y) has the value false.
- 4 5 **Result Characteristics.** Real with the kind type parameter of whichever argument has the greater preci-
5 sion.
- 6 6 **Result Value.** This function computes the remainder operation specified in ISO/IEC 60559:2020.
- 7 7 The result value when X and Y are finite, and Y is nonzero, regardless of the rounding mode, shall be exactly
8 $X - Y \cdot N$, where N is the integer nearest to the exact value X/Y ; whenever $|N - X/Y| = \frac{1}{2}$, N shall be even. If
9 the result value is zero, the sign shall be that of X.
- 10 8 When X is finite and Y is infinite, the result value is X. If Y is zero or X is infinite, and neither is a NaN,
11 the IEEE_INVALID exception shall occur; if IEEE_SUPPORT_NAN(X+Y) is true, the result is a NaN. If X is
12 subnormal and Y is infinite, the IEEE_UNDERFLOW exception shall occur. No exception shall signal if X is
13 finite and normal, and Y is infinite.
- 14 9 **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
15 of IEEE_REM (5.0, 2.0) is 1.0.

16 17.11.36 IEEE_RINT (X [, ROUND])

- 17 1 **Description.** Round to integer.
- 18 2 **Class.** Elemental function.
- 19 3 **Arguments.**
- 20 X shall be of type real.
- 21 ROUND (optional) shall be of type IEEE_ROUND_TYPE.
- 22 4 **Restriction.** IEEE_RINT (X) shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the value false.
- 23 5 **Result Characteristics.** Same as X.
- 24 6 **Result Value.** If ROUND is present, the value of the result is the value of X rounded to an integer according
25 to the mode specified by ROUND; this is the ISO/IEC/IEEE 60559:2020 operation roundToIntegral{round-
26 ing}. Otherwise, the value of the result is that specified for the operation roundToIntegralExact in ISO/IEC
27 60559:2020; this is the value of X rounded to an integer according to the rounding mode. If the result has
28 the value zero, the sign is that of X.
- 29 7 **Examples.** If the rounding mode is round to nearest, the value of IEEE_RINT (1.1) is 1.0. The value of
30 IEEE_RINT (1.1, IEEE_UP) is 2.0.

31 17.11.37 IEEE_SCALB (X, I)

- 32 1 **Description.** $X \times 2^I$.
- 33 2 **Class.** Elemental function.
- 34 3 **Arguments.**
- 35 X shall be of type real.

I shall be of type integer or of type real with the same kind type parameter as X.

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.

The value of the result shall conform to the scaleB operation of ISO/IEC 60559:2020.

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 $|X \times 2^I|$ 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.

17.11.38 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 60559:2020 floating-point format with decimal precision, as returned by the intrinsic function PRECISION, of at least P digits, a decimal exponent range, as returned by the intrinsic function RANGE, of at least R, and a radix, as returned by the intrinsic function RADIX, of RADIX, if such a kind type parameter is available on the processor.

Otherwise, the result is -1 if the processor supports an IEEE real type with radix RADIX and exponent range of at least R but not with precision of at least P, -2 if the processor supports an IEEE real type with radix RADIX and precision of at least P but not with exponent range of at least R, -3 if the processor supports an IEEE real type with radix RADIX but with neither precision of at least P nor exponent range of at least R, -4 if the processor supports an IEEE real type with radix RADIX and either precision of at least P or exponent range of at least R but not both together, and -5 if the processor supports no IEEE real type with radix RADIX.

If more than one kind type parameter value meets the criteria, the value returned is the one with the smallest decimal precision, unless there are several such values, in which case the smallest of these kind values is returned.

Example. IEEE_SELECTED_REAL_KIND (6, 30) has the value **KIND** (0.0) on a machine that supports ISO/IEC/IEEE 60559:2020 binary32 arithmetic for its default real approximation method.

17.11.39 IEEE_SET_FLAG (FLAG, FLAG_VALUE)

Class. **Simple** subroutine.

2 Arguments.

FLAG shall be a scalar or array of type **IEEE_FLAG_TYPE**. If a value of **FLAG** is IEEE_INVALID, IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_UNDERFLOW, or IEEE_INEXACT, the corresponding exception flag is assigned a value. No two elements of **FLAG** shall have the same value.

FLAG_VALUE shall be a logical scalar or array. It shall be **conformable** with **FLAG**. If an element has the value true, the corresponding flag is set to be signaling; otherwise, the flag is set to be quiet.

Example. CALL IEEE_SET_FLAG (IEEE_OVERFLOW, **.TRUE.**) sets the IEEE_OVERFLOW flag to be signaling.

17.11.40 IEEE_SET_HALTING_MODE (FLAG, HALTING)

Description. Set a **halting mode**.

Class. **Simple** 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.

Restriction. IEEE_SET_HALTING_MODE (FLAG, HALTING) shall not be invoked if IEEE_SUPPORT_HALTING (FLAG) has the value false.

Example. CALL IEEE_SET_HALTING_MODE (IEEE_DIVIDE_BY_ZERO, **.TRUE.**) causes halting after a divide-by-zero exception.

17.11.41 IEEE_SET_MODES (MODES)

Description. Set floating-point modes.

Class. **Simple** subroutine.

Argument. **MODES** shall be a scalar of type **IEEE_MODES_TYPE**. Its value shall be one that was assigned by a previous invocation of **IEEE_GET_MODES** to its **MODES** argument. The floating-point modes (17.7) are restored to the state at that invocation.

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
...
```

```

1      CALL IEEE_GET_MODES (SAVE_MODES) ! Save all modes.
2      CALL IEEE_SET_ROUNDING_MODE (IEEE_TO_ZERO))
3      CALL IEEE_SET_UNDERFLOW_MODE (GRADUAL=.FALSE.)
4      ... ! calculation with abrupt round-to-zero.
5      CALL IEEE_SET_MODES (SAVE_MODES) ! Restore all modes.

```

17.11.42 IEEE_SET_ROUNDING_MODE (ROUND_VALUE [, RADIX])

1 **Description.** Set rounding mode.

2 **Class.** [Simple](#) subroutine.

3 **Arguments.**

ROUND_VALUE shall be a scalar of type [IEEE_ROUND_TYPE](#). It specifies the rounding mode to be set.

RADIX (optional) shall be an integer scalar with the value two or ten. If RADIX is present with the value ten, the rounding mode set is the decimal rounding mode; otherwise it is the binary rounding mode.

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:

```

21      USE, INTRINSIC :: IEEE_ARITHMETIC
22      TYPE (IEEE_ROUND_TYPE) ROUND_VALUE
23      ...
24      CALL IEEE_GET_ROUNDING_MODE (ROUND_VALUE) ! Store the rounding mode
25      CALL IEEE_SET_ROUNDING_MODE (IEEE_NEAREST)
26      ... ! calculation with round to nearest
27      CALL IEEE_SET_ROUNDING_MODE (ROUND_VALUE) ! Restore the rounding mode

```

17.11.43 IEEE_SET_STATUS (STATUS_VALUE)

1 **Description.** Restore floating-point status.

2 **Class.** [Simple](#) 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 ([17.7](#) 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:

```

36      USE, INTRINSIC :: IEEE_EXCEPTIONS
37      TYPE (IEEE_STATUS_TYPE) STATUS_VALUE
38      ...
39      CALL IEEE_GET_STATUS (STATUS_VALUE) ! Store the flags
40      CALL IEEE_SET_FLAG (IEEE_ALL, .FALSE.) ! Set them quiet
41      ... ! calculation involving exception handling
42      CALL IEEE_SET_STATUS (STATUS_VALUE) ! Restore the flags

```

17.11.44 IEEE_SET_UNDERFLOW_MODE (GRADUAL)

Description. Set [underflow mode](#).

Class. [Simple](#) subroutine.

Argument. GRADUAL shall be a logical scalar. If it is true, the [underflow mode](#) is set to gradual underflow. If it is false, the [underflow mode](#) is set to abrupt underflow.

Restriction. IEEE_SET_UNDERFLOW_MODE shall not be invoked unless IEEE_SUPPORT_UNDERFLOW_CONTROL (X) is true for some X.

Example. To perform some calculations with abrupt underflow and then restore the previous mode:

```

USE, INTRINSIC :: IEEE_ARITHMETIC
LOGICAL SAVE_UNDERFLOW_MODE
...
CALL IEEE_GET_UNDERFLOW_MODE (SAVE_UNDERFLOW_MODE)
CALL IEEE_SET_UNDERFLOW_MODE (GRADUAL=.FALSE.)
... ! Perform some calculations with abrupt underflow
CALL IEEE_SET_UNDERFLOW_MODE (SAVE_UNDERFLOW_MODE)

```

17.11.45 IEEE_SIGNALING_EQ (A, B)

Description. Signaling compares equal.

Class. [Elemental](#) function.

Arguments.

A shall be of type real.

B shall be of the same type and kind type parameter as A.

Restriction. IEEE_SIGNALING_EQ (A, B) shall not be invoked if [IEEE_SUPPORT_DATATYPE](#) (A) has the value false.

Result Characteristics. Default logical.

Result Value. The result has the value specified for the compareSignalingEqual operation in ISO/IEC 60559:2020; that is, it is true if and only if A compares equal to B. If A or B is a NaN, the result will be false and IEEE_INVALID signals; otherwise, no exception is signaled.

Example. IEEE_SIGNALING_EQ (1.0, [IEEE_VALUE](#) (1.0, IEEE_QUIET_NAN)) has the value false and signals IEEE_INVALID.

17.11.46 IEEE_SIGNALING_GE (A, B)

Description. Signaling compares greater than or equal.

Class. [Elemental](#) function.

Arguments.

A shall be of type real.

B shall be of the same type and kind type parameter as A.

1 4 **Restriction.** IEEE_SIGNALING_GE (A, B) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the
2 value false.

3 5 **Result Characteristics.** Default logical.

4 6 **Result Value.** The result has the value specified for the compareSignalingGreaterEqual operation in
5 ISO/IEC 60559:2020; that is, it is true if and only if A compares greater than or equal to B. If A or B is a
6 NaN, the result will be false and IEEE_INVALID signals; otherwise, no exception is signaled.

7 7 **Example.** IEEE_SIGNALING_GE (1.0, IEEE_VALUE (1.0, IEEE_QUIET_NAN)) has the value false and signals
8 IEEE_INVALID.

9 17.11.47 IEEE_SIGNALING_GT (A, B)

10 1 **Description.** Signaling compares greater than.

11 2 **Class.** Elemental function.

12 3 **Arguments.**

13 A shall be of type real.

14 B shall be of the same type and kind type parameter as A.

15 4 **Restriction.** IEEE_SIGNALING_GT (A, B) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the
16 value false.

17 5 **Result Characteristics.** Default logical.

18 6 **Result Value.** The result has the value specified for the compareSignalingGreater operation in ISO/IEC
19 60559:2020; 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
20 false and IEEE_INVALID signals; otherwise, no exception is signaled.

21 7 **Example.** IEEE_SIGNALING_GT (1.0, IEEE_VALUE (1.0, IEEE_QUIET_NAN)) has the value false and signals
22 IEEE_INVALID.

23 17.11.48 IEEE_SIGNALING_LE (A, B)

24 1 **Description.** Signaling compares less than or equal.

25 2 **Class.** Elemental function.

26 3 **Arguments.**

27 A shall be of type real.

28 B shall be of the same type and kind type parameter as A.

29 4 **Restriction.** IEEE_SIGNALING_LE (A, B) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the value
30 false.

31 5 **Result Characteristics.** Default logical.

32 6 **Result Value.** The result has the value specified for the compareSignalingLessEqual operation in ISO/IEC
33 60559:2020; 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
34 will be false and IEEE_INVALID signals; otherwise, no exception is signaled.

1 7 **Example.** IEEE_SIGNALING_LE (1.0, IEEE_VALUE (1.0, IEEE_QUIET_NAN)) has the value false and signals
2 IEEE_INVALID.

3 17.11.49 IEEE_SIGNALING_LT (A, B)

4 1 **Description.** Signaling compares less than.

5 2 **Class.** Elemental function.

6 3 **Arguments.**

7 A shall be of type real.

8 B shall be of the same type and kind type parameter as A.

9 4 **Restriction.** IEEE_SIGNALING_LT (A, B) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the value
10 false.

11 5 **Result Characteristics.** Default logical.

12 6 **Result Value.** The result has the value specified for the compareSignalingLess operation in ISO/IEC
13 60559:2020; 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
14 false and IEEE_INVALID signals; otherwise, no exception is signaled.

15 7 **Example.** IEEE_SIGNALING_LT (1.0, IEEE_VALUE (1.0, IEEE_QUIET_NAN)) has the value false and signals
16 IEEE_INVALID.

17 17.11.50 IEEE_SIGNALING_NE (A, B)

18 1 **Description.** Signaling compares not equal.

19 2 **Class.** Elemental function.

20 3 **Arguments.**

21 A shall be of type real.

22 B shall be of the same type and kind type parameter as A.

23 4 **Restriction.** IEEE_SIGNALING_NE (A, B) shall not be invoked if IEEE_SUPPORT_DATATYPE (A) has the
24 value false.

25 5 **Result Characteristics.** Default logical.

26 6 **Result Value.** The result has the value specified for the compareSignalingNotEqual operation in ISO/IEC
27 60559:2020; 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
28 true and IEEE_INVALID signals; otherwise, no exception is signaled.

29 7 **Example.** IEEE_SIGNALING_NE (1.0, IEEE_VALUE (1.0, IEEE_QUIET_NAN)) has the value true and signals
30 IEEE_INVALID.

31 17.11.51 IEEE_SIGNBIT (X)

32 1 **Description.** Test sign bit.

33 2 **Class.** Elemental function.

34 3 **Argument.** X shall be of type real.

1 4 **Restriction.** IEEE_SIGNBIT (X) shall not be invoked if IEEE_SUPPORT_DATATYPE (X) has the value false.

2 5 **Result Characteristics.** Default logical.

3 6 **Result Value.** The result has the value specified for the isSignMinus operation in ISO/IEC 60559:2020; that
4 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.

5 7 **Example.** IEEE_SIGNBIT (−1.0) has the value true.

6 17.11.52 IEEE_SUPPORT_DATATYPE () or IEEE_SUPPORT_DATATYPE (X)

7 1 **Description.** Query IEEE arithmetic support.

8 2 **Class.** Inquiry function.

9 3 **Argument.** X shall be of type real. It may be a scalar or an array.

10 4 **Result Characteristics.** Default logical scalar.

11 5 **Result Value.** The result has the value true if the processor supports IEEE arithmetic for all reals (X does
12 not appear) or for real variables of the same kind type parameter as X; otherwise, it has the value false.
13 Here, support is as defined in the first paragraph of 17.9.

14 6 **Example.** If default real kind conforms to ISO/IEC 60559:2020 except that underflow values flush to zero
15 instead of being subnormal, IEEE_SUPPORT_DATATYPE (1.0) has the value true.

16 17.11.53 IEEE_SUPPORT_DENORMAL () or IEEE_SUPPORT_DENORMAL (X)

17 1 **Description.** Query subnormal number support.

18 2 **Class.** Inquiry function.

19 3 **Argument.** X shall be of type real. It may be a scalar or an array.

20 4 **Result Characteristics.** Default logical scalar.

21 5 **Result Value.**

22 *Case (i):* IEEE_SUPPORT_DENORMAL (X) has the value true if IEEE_SUPPORT_DATATYPE (X) has the
23 value true and the processor supports arithmetic operations and assignments with subnormal
24 numbers (biased exponent $e = 0$ and fraction $f \neq 0$, see ISO/IEC 60559:2020, 3.2) for real
25 variables of the same kind type parameter as X; otherwise, it has the value false.

26 *Case (ii):* IEEE_SUPPORT_DENORMAL () has the value true if IEEE_SUPPORT_DENORMAL (X) has the
27 value true for all real X; otherwise, it has the value false.

28 6 **Example.** IEEE_SUPPORT_DENORMAL (X) has the value true if the processor supports subnormal values
29 for X.

NOTE

A reference to IEEE_SUPPORT_DENORMAL will have the same result value as a reference to IEEE_SUPPORT_SUB-NORMAL with the same argument list.

17.11.54 IEEE_SUPPORT_DIVIDE () or IEEE_SUPPORT_DIVIDE (X)

Description. Query IEEE division support.

Class. [Inquiry function](#).

Argument. X shall be of type real. It may be a scalar or an array.

Result Characteristics. Default logical scalar.

Result Value.

Case (i): IEEE_SUPPORT_DIVIDE (X) has the value true if the processor supports division with the accuracy specified by ISO/IEC 60559:2020 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.

Example. IEEE_SUPPORT_DIVIDE (X) has the value true if division of operands with the same kind as X conforms to ISO/IEC 60559:2020.

17.11.55 IEEE_SUPPORT_FLAG (FLAG) or IEEE_SUPPORT_FLAG (FLAG, X)

Description. Query exception support.

Class. [Transformational function](#).

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.

Result Characteristics. Default logical scalar.

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.

Example. IEEE_SUPPORT_FLAG (IEEE_INEXACT) has the value true if the processor supports the inexact exception.

17.11.56 IEEE_SUPPORT_HALTING (FLAG)

Description. Query [halting mode](#) support.

Class. [Transformational function](#).

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.

Result Characteristics. Default logical scalar.

Result Value. The result has the value true if the processor supports the ability to control during program execution whether to abort or continue execution after the exception specified by FLAG; otherwise, it has the value false. Support includes the ability to change the mode by CALL [IEEE_SET_HALTING_MODE](#) (FLAG).

Example. IEEE_SUPPORT_HALTING (IEEE_OVERFLOW) has the value true if the processor supports control of [halting](#) after an overflow.

17.11.57 IEEE_SUPPORT_INF () or IEEE_SUPPORT_INF (X)

Description. Query IEEE infinity support.

Class. [Inquiry function](#).

Argument. X shall be of type real. It may be a scalar or an array.

Result Characteristics. Default logical scalar.

Result Value.

Case (i): IEEE_SUPPORT_INF (X) has the value true if the processor supports IEEE infinities (positive and negative) for real variables of the same kind type parameter as X; otherwise, it has the value false.

Case (ii): IEEE_SUPPORT_INF () has the value true if IEEE_SUPPORT_INF (X) has the value true for all real X; otherwise, it has the value false.

Example. IEEE_SUPPORT_INF (X) has the value true if the processor supports IEEE infinities for X.

17.11.58 IEEE_SUPPORT_IO () or IEEE_SUPPORT_IO (X)

Description. Query IEEE formatting support.

Class. [Inquiry function](#).

Argument. X shall be of type real. It may be a scalar or an array.

Result Characteristics. Default logical scalar.

Result Value.

Case (i): IEEE_SUPPORT_IO (X) has the value true if base conversion during formatted input/output ([12.5.6.17](#), [12.6.2.14](#), [13.7.2.3.8](#)) conforms to ISO/IEC 60559:2020 for the modes UP, DOWN, ZERO, and NEAREST for real variables of the same kind type parameter as X; otherwise, it has the value false.

Case (ii): IEEE_SUPPORT_IO () has the value true if IEEE_SUPPORT_IO (X) has the value true for all real X; otherwise, it has the value false.

Example. IEEE_SUPPORT_IO (X) has the value true if formatted input/output base conversions conform to ISO/IEC 60559:2020.

17.11.59 IEEE_SUPPORT_NAN () or IEEE_SUPPORT_NAN (X)

Description. Query [IEEE NaN](#) support.

Class. [Inquiry function](#).

1 3 **Argument.** X shall be of type real. It may be a scalar or an array.

2 4 **Result Characteristics.** Default logical scalar.

3 5 **Result Value.**

4 *Case (i):* IEEE_SUPPORT_NAN (X) has the value true if the processor supports IEEE NaNs for real vari-
5 ables of the same kind type parameter as X; otherwise, it has the value false.

6 *Case (ii):* IEEE_SUPPORT_NAN () has the value true if IEEE_SUPPORT_NAN (X) has the value true for all
7 real X; otherwise, it has the value false.

8 6 **Example.** IEEE_SUPPORT_NAN (X) has the value true if the processor supports IEEE NaNs for X.

9 17.11.60 IEEE_SUPPORT_ROUNDING (ROUND_VALUE) or IEEE_SUPPORT_ROUNDING (ROUND_VALUE, X)

10 1 **Description.** Query IEEE rounding support.

11 2 **Class.** Transformational function.

12 3 **Arguments.**

13 ROUND_VALUE shall be of type IEEE_ROUND_TYPE.

14 X shall be of type real. It may be a scalar or an array.

15 4 **Result Characteristics.** Default logical scalar.

16 5 **Result Value.**

17 *Case (i):* IEEE_SUPPORT_ROUNDING (ROUND_VALUE, X) has the value true if the processor supports
18 the rounding mode defined by ROUND_VALUE for real variables of the same kind type para-
19 meter as X; otherwise, it has the value false. Support includes the ability to change the mode
20 by CALL IEEE_SET_ROUNDING_MODE (ROUND_VALUE).

21 *Case (ii):* IEEE_SUPPORT_ROUNDING (ROUND_VALUE) has the value true if IEEE_SUPPORT_-
22 ROUNDING (ROUND_VALUE, X) has the value true for all real X; otherwise, it has the
23 value false.

24 6 **Example.** IEEE_SUPPORT_ROUNDING (IEEE_TO_ZERO) has the value true if the processor supports round-
25 ing to zero for all reals.

26 17.11.61 IEEE_SUPPORT_SQRT () or IEEE_SUPPORT_SQRT (X)

27 1 **Description.** Query IEEE square root support.

28 2 **Class.** Inquiry function.

29 3 **Argument.** X shall be of type real. It may be a scalar or an array.

30 4 **Result Characteristics.** Default logical scalar.

31 5 **Result Value.**

32 *Case (i):* IEEE_SUPPORT_SQRT (X) has the value true if the intrinsic function SQRT conforms to ISO/IEC
33 60559:2020 for real variables of the same kind type parameter as X; otherwise, it has the value
34 false.

1 *Case (ii):* IEEE_SUPPORT_SQRT () has the value true if IEEE_SUPPORT_SQRT (X) has the value true for
2 all real X; otherwise, it has the value false.

3 6 **Example.** If IEEE_SUPPORT_SQRT (1.0) has the value true, SQRT (−0.0) will have the value −0.0.

4 17.11.62 IEEE_SUPPORT_STANDARD () or IEEE_SUPPORT_STANDARD (X)

5 1 **Description.** Query IEEE standard support.

6 2 **Class.** Inquiry function.

7 3 **Argument.** X shall be of type real. It may be a scalar or an array.

8 4 **Result Characteristics.** Default logical scalar.

9 5 **Result Value.**

10 *Case (i):* IEEE_SUPPORT_STANDARD (X) has the value true if the results of all the functions IEEE_
11 SUPPORT_DATATYPE (X), IEEE_SUPPORT_DIVIDE (X), IEEE_SUPPORT_FLAG (FLAG, X) for
12 valid FLAG, IEEE_SUPPORT_HALTING (FLAG) for valid FLAG, IEEE_SUPPORT_INF (X), IEEE_
13 SUPPORT_NAN (X), IEEE_SUPPORT_ROUNDING (ROUND_VALUE, X) for valid ROUND_VALUE,
14 IEEE_SUPPORT_SQRT (X), and IEEE_SUPPORT_SUBNORMAL (X) are all true; otherwise, it has
15 the value false.

16 *Case (ii):* IEEE_SUPPORT_STANDARD () has the value true if IEEE_SUPPORT_STANDARD (X) has the
17 value true for all real X; otherwise, it has the value false.

18 6 **Example.** IEEE_SUPPORT_STANDARD () has the value false if some but not all kinds of reals conform to
19 ISO/IEC 60559:2020.

20 17.11.63 IEEE_SUPPORT_SUBNORMAL () or IEEE_SUPPORT_SUBNORMAL (X)

21 1 **Description.** Query subnormal number support.

22 2 **Class.** Inquiry function.

23 3 **Argument.** X shall be of type real. It may be a scalar or an array.

24 4 **Result Characteristics.** Default logical scalar.

25 5 **Result Value.**

26 *Case (i):* IEEE_SUPPORT_SUBNORMAL (X) has the value true if IEEE_SUPPORT_DATATYPE (X) has the
27 value true and the processor supports arithmetic operations and assignments with subnormal
28 numbers (biased exponent $e = 0$ and fraction $f \neq 0$, see ISO/IEC 60559:2020, 3.2) for real
29 variables of the same kind type parameter as X; otherwise, it has the value false.

30 *Case (ii):* IEEE_SUPPORT_SUBNORMAL () has the value true if IEEE_SUPPORT_SUBNORMAL (X) has the
31 value true for all real X; otherwise, it has the value false.

32 6 **Example.** IEEE_SUPPORT_SUBNORMAL (X) has the value true if the processor supports subnormal values
33 for X.

NOTE

The subnormal numbers are not included in the 16.4 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

NOTE (cont.)

operation causes IEEE_UNDERFLOW to signal unless the result is exact. IEEE_SUPPORT_SUBNORMAL (X) is false if the processor never returns a subnormal number as the result of an arithmetic operation.

17.11.64 IEEE_SUPPORT_UNDERFLOW_CONTROL () or IEEE_SUPPORT_UNDERFLOW_CONTROL (X)

Description. Query [underflow](#) control support.

Class. [Inquiry function](#).

Argument. X shall be of type real. It may be a scalar or an array.

Result Characteristics. Default logical scalar.

Result Value.

Case (i): IEEE_SUPPORT_UNDERFLOW_CONTROL (X) has the value true if the processor supports control of the [underflow mode](#) for floating-point calculations with the same type as X, and false otherwise.

Case (ii): IEEE_SUPPORT_UNDERFLOW_CONTROL () has the value true if the processor supports control of the [underflow mode](#) for all floating-point calculations, and false otherwise.

Example. IEEE_SUPPORT_UNDERFLOW_CONTROL (2.5) has the value true if the processor supports [underflow mode](#) control for default real calculations.

17.11.65 IEEE_UNORDERED (X, Y)

Description. Whether two values are unordered.

Class. [Elemental function](#).

Arguments. The arguments shall be of type real.

Restriction. IEEE_UNORDERED (X, Y) shall not be invoked if IEEE_SUPPORT_DATATYPE (X) or IEEE_SUPPORT_DATATYPE (Y) has the value false.

Result Characteristics. Default logical.

Result Value. The result has the value true if X or Y is a NaN or both are NaNs; otherwise, it has the value false. If X or Y is a signaling NaN, IEEE_INVALID may signal.

Example. IEEE_UNORDERED (0.0, [SQRT](#) (−1.0)) has the value true if IEEE_SUPPORT_SQRT (1.0) has the value true.

17.11.66 IEEE_VALUE (X, CLASS)

Description. Return number in a class.

Class. [Elemental function](#).

Arguments.

X shall be of type real.

- 1 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.
- 7 4 **Restriction.** IEEE_VALUE (X, CLASS) shall not be invoked if [IEEE_SUPPORT_DATATYPE](#) (X) has the value false.
- 9 5 **Result Characteristics.** Same as X.
- 10 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.
- 13 7 **Example.** IEEE_VALUE (1.0, IEEE_NEGATIVE_INF) has the value $-\infty$.
- 14 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

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.

16 **17.12 Examples****NOTE 1**

```

MODULE DOT
  ! Module for dot product of two real arrays of rank 1.
  ! The caller needs to ensure that exceptions do not cause halting.
  USE, INTRINSIC :: IEEE_EXCEPTIONS
  LOGICAL :: MATRIX_ERROR = .FALSE.
  INTERFACE OPERATOR(.dot.)
    MODULE PROCEDURE MULT
  END INTERFACE
CONTAINS
  REAL FUNCTION MULT (A, B)
    REAL, INTENT (IN) :: A(:), B(:)
    INTEGER I
    LOGICAL OVERFLOW
    IF (SIZE(A) /= SIZE(B)) THEN
      MATRIX_ERROR = .TRUE.
      RETURN
    END IF
    ! The processor ensures that IEEE_OVERFLOW is quiet.
    MULT = 0.0
    DO I = 1, SIZE (A)
      MULT = MULT + A(I)*B(I)
    END DO
    CALL IEEE_GET_FLAG (IEEE_OVERFLOW, OVERFLOW)
    IF (OVERFLOW) MATRIX_ERROR = .TRUE.
  END FUNCTION MULT

```

NOTE 1 (cont.)

```
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 2

```
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)
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 3

```
USE, INTRINSIC :: IEEE_EXCEPTIONS
LOGICAL FLAG_VALUE
...
CALL IEEE_SET_HALTING_MODE (IEEE_OVERFLOW, .FALSE.)
! First try a fast algorithm for inverting a matrix.
CALL IEEE_SET_FLAG (IEEE_OVERFLOW, .FALSE.)
DO K = 1, N
...
CALL IEEE_GET_FLAG (IEEE_OVERFLOW, FLAG_VALUE)
IF (FLAG_VALUE) EXIT
END DO
IF (FLAG_VALUE) THEN
! Alternative code which knows that K-1 steps have executed normally.
...
END IF
```

Here the code for matrix inversion is in line and the transfer is made more precise by adding extra tests of the flag.

18 Interoperability with C

18.1 General

- 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 ISO/IEC 9899:2018, 6.7.6.3, 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 ISO/IEC 9899:2018, 6.2.2.
- 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 ([7.5](#)) and interoperable enumerations ([7.6.1](#)) 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.
- 4 The conditions under which a Fortran entity is [interoperable](#) are defined in [18.3](#). 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

A Fortran entity can be interoperable with more than one C entity.

18.2 The ISO_C_BINDING intrinsic module

18.2.1 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`, `C_NULL_FUNPTR`, and those with names listed in the first column of Table [18.1](#) and the second column of Table [18.2](#), the types `C_PTR` and `C_FUNPTR`, and the procedures in [18.2.3](#). A processor may provide other public entities in the ISO_C_BINDING intrinsic module in addition to those listed here.

18.2.2 Named constants and derived types in the module

- 1 The entities listed in the second column of Table [18.2](#) 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

- 1 C_SHORT, C_LONG, C_LONG_LONG, C_SIGNED_CHAR, C_SIZE_T, C_INT8_T, C_INT16_T,
 2 C_INT32_T, C_INT64_T, C_INT_LEAST8_T, C_INT_LEAST16_T, C_INT_LEAST32_T, C_INT_LEAST64_T, C_INT_
 3 FAST8_T, C_INT_FAST16_T, C_INT_FAST32_T, C_INT_FAST64_T, C_INTMAX_T, C_INTPTR_T, and C_PTRDIFF_T
 4 shall each be a valid value for an integer **kind type parameter** on the processor or shall be -1 if the **com-**
 5 **panion processor** (5.5.7) defines the corresponding C type and there is no interoperating Fortran processor
 6 kind, or -2 if the **companion processor** does not define the corresponding C type.
- 7 4 The values of C_FLOAT, C_DOUBLE, and C_LONG_DOUBLE shall each be a valid value for a real **kind type**
 8 **parameter** on the processor or shall be -1 if the **companion processor**'s type does not have a precision equal
 9 to the precision of any of the Fortran processor's real kinds, -2 if the **companion processor**'s type does not
 10 have a range equal to the range of any of the Fortran processor's real kinds, -3 if the **companion processor**'s
 11 type has neither the precision nor range of any of the Fortran processor's real kinds, and equal to -4 if
 12 there is no interoperating Fortran processor kind for other reasons. The values of C_FLOAT_COMPLEX, C_
 13 DOUBLE_COMPLEX, and C_LONG_DOUBLE_COMPLEX shall be the same as those of C_FLOAT, C_DOUBLE,
 14 and C_LONG_DOUBLE, respectively.
- 15 5 The value of C_BOOL shall be a valid value for a logical kind parameter on the processor or shall be -1 .
- 16 6 The value of C_CHAR shall be a valid value for a character **kind type parameter** on the processor or shall be
 17 -1 . If the value of C_CHAR is nonnegative, the character kind specified is the C character kind; otherwise,
 18 there is no C character kind.
- 19 7 The following entities shall be **named constants** of type character with a length parameter of one. The
 20 kind parameter value shall be equal to the value of C_CHAR unless C_CHAR = -1 , in which case the kind
 21 parameter value shall be the same as for default kind. The values of these constants are specified in Table
 22 18.1. In the case that C_CHAR $\neq -1$ the value is specified using C syntax. The semantics of these values are
 23 explained in ISO/IEC 9899:2018, 5.2.1 and 5.2.2.

Table 18.1 — Names of C characters with special semantics

Name	C definition	Value	
		C_CHAR = -1	C_CHAR $\neq -1$
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'

- 24 8 The entities **C_PTR** and **C_FUNPTR** are described in 18.3.2.
- 25 9 The entity C_NULL_PTR shall be a **named constant** of type **C_PTR**. The value of C_NULL_PTR shall be the
 26 same as the value NULL in C. The entity C_NULL_FUNPTR shall be a **named constant** of type **C_FUNPTR**.
 27 The value of C_NULL_FUNPTR shall be that of a null pointer to a function in C.

NOTE

The value of NEW_LINE (C_NEW_LINE) is C_NEW_LINE (16.9.150).

18.2.3 Procedures in the module

18.2.3.1 General

In the detailed descriptions below, procedure names are generic and not specific. The `C_F_POINTER`, `C_F_PROCPOINTER`, and `C_F_STRPOINTER` subroutines are impure; all other procedures in the module are simple.

18.2.3.2 C_ASSOCIATED (C_PTR_1 [, C_PTR_2])

Description. Query C pointer status.

Class. Transformational function.

Arguments.

`C_PTR_1` shall be a scalar of type `C_PTR` or `C_FUNPTR`.

`C_PTR_2` (optional) shall be a scalar of the same type as `C_PTR_1`.

Result Characteristics. Default logical scalar.

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 ISO/IEC 9899:2018, 6.3.2.3 and 6.5.9, and false otherwise.

Examples.

Case (i): If variable `P` of type `C_PTR` has been assigned the value of `C_NULL_PTR`, the value of `C_ASSOCIATED (P)` is false.

Case (ii): For the interoperable variable `REAL (C_DOUBLE), TARGET, BIND (C) :: X`, if variable `P` of type `C_PTR` has been assigned the address of `X`, perhaps by a C function that used “&x”, the value of `C_ASSOCIATED (P, C_LOC (X))` is true.

18.2.3.3 C_F_POINTER (CPTR, FPTR [, SHAPE, LOWER])

Description. Associate a data pointer with the target of a C pointer and specify its shape.

Class. Subroutine.

Arguments.

`CPTR` shall be a scalar of type `C_PTR`. It is an **INTENT (IN)** argument. Its value shall be

- the C address of an interoperable data entity,
- the result of a reference to `C_LOC` with a noninteroperable argument, or
- the C address of a storage sequence that is not in use by any other Fortran entity.

The value of `CPTR` shall not be the C address of a Fortran variable that does not have the **TARGET** attribute.

`FPTR` shall be a pointer, shall not have a deferred type parameter, and shall not be a coindexed object. It is an **INTENT (OUT)** argument. If `FPTR` is an array, its shape is specified by `SHAPE`; the lower bounds are specified by `LOWER` if it is present, otherwise each lower bound is equal to 1.

Case (i): If the value of CPTR is the **C address** of an **interoperable** data entity, FPTR shall be a **data pointer** with type and **type parameter** values **interoperable** with the type of the entity. If the target T of CPTR is scalar, FPTR becomes **pointer associated** with T; if FPTR is an array, SHAPE shall specify a size of 1. If T is an array, and FPTR is scalar, FPTR becomes associated with the first element of T. If both T and FPTR are arrays, SHAPE shall specify a size that is less than or equal to the size of T, and FPTR becomes associated with the first **PRODUCT** (SHAPE) elements of T (this could be the entirety of T).

Case (ii): If the value of CPTR is the result of a reference to C_LOC with a noninteroperable **effective argument** X, FPTR shall be a nonpolymorphic pointer with the same type and type parameters as X. In this case, X shall not have been deallocated or have become undefined due to execution of a **RETURN** or **END** statement since the reference. If X is scalar, FPTR becomes **pointer associated** with X; if FPTR is an array, SHAPE shall specify a size of 1. If X is an array and FPTR is scalar, FPTR becomes associated with the first element of X. If both X and FPTR are arrays, SHAPE shall specify a size that is less than or equal to the size of X, and FPTR becomes associated with the first **PRODUCT** (SHAPE) elements of X (this could be the entirety of X).

Case (iii): If the value of CPTR is the **C address** of a **storage sequence** that is not in use by any other Fortran entity, FPTR becomes associated with that **storage sequence**. 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.

LOWER (optional) shall be a rank-one integer array. It is an **INTENT (IN)** argument. It shall not be present if SHAPE is not present. If LOWER is present, 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(0:2)
Type(C_ptr) :: xloc
xloc = C_Loc (x)
...
Type(t), Pointer :: y(:)
Call C_F_Pointer (xloc, y, [3], [0])
```

Case (iii):

```
void *getmem (int nbits)
{
    return malloc ((nbits+CHAR_BIT-1)/CHAR_BIT);
}
```

```

1      ! Assume interface to "getmem" is available,
2      ! and there is a derived type "mytype" accessible.
3      Type(mytype), Pointer :: x
4      Call C_F_Pointer (getmem (Storage_Size (x)), x)

```

Case (iv): The following statements illustrate the use of C_F_POINTER when the pointer to be set has a **deferred type parameter**:

```

7      Character(42), Pointer :: C1
8      Character(:), Pointer :: C2
9      Call C_F_Pointer (CPTR, C1)
10     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

In the case of associating FPTR with a **storage sequence**, there might be processor-dependent requirements such as alignment of the memory address or placement in memory.

18.2.3.4 C_F_PROCPTR (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.

4 **Example.**

The following C code provides a function, `dispatch`, that returns a C function pointer to the C library cube root function:

```

28     #include <math.h>
29     typedef double (*simplefun) (double);
30
31     simplefun dispatch (void) {
32         return &cbirt;
33     }

```

The following Fortran interface interoperates with `dispatch`:

```

35     Interface
36     Type(C_FUNPTR) Function dispatch () Bind(C)
37     Use Iso_C_Binding, Only: C_FUNPTR
38     End Function dispatch
39     End Interface

```

With the abstract interface **SIMPLE_FUNCTION** (analogous to `simplefun`), a procedure pointer suitable for referring to the C library function `cbirt` can be created:

```

1      Abstract Interface
2      Real (C_double) Function simple_function (x) Bind(C)
3      Use Iso_C_Binding, Only: C_double
4      Real (C_double), Value :: x
5      End Function simple_function
6      End Interface
7      Procedure (simple_function), Pointer :: psimp

```

8 Once the procedure pointer is associated, it can be used to invoke `cbrt`:

```

9      Call C_F_Procpointer (dispatch (), psimp)
10     Write (*,*) psimp (4.5_C_double)

```

NOTE

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 ISO/IEC 9899:2018, 6.2.5.

11 18.2.3.5 C_F_STRPOINTER (CSTRARRAY, FSTRPTR [, NCHARS]) or C_F_STRPOINTER (CSTRPTR, FSTRPTR [, NCHARS])

12 1 **Description.** Associate a character pointer with a C string.

13 2 **Class.** Subroutine.

14 3 **Arguments.**

15 CSTRARRAY shall be a rank one character array of kind `C_CHAR`, with a length type parameter equal to
16 one. It is an **INTENT (IN)** argument. Its **actual argument** shall be **simply contiguous** and have
17 the **TARGET attribute**.

18 CSTRPTR shall be a scalar of type `C_PTR`. It is an **INTENT (IN)** argument. Its value shall be the **C address**
19 of a contiguous array S of NCHARS characters. Its value shall not be the **C address** of a Fortran
20 variable that does not have the **TARGET attribute**.

21 FSTRPTR shall be a scalar deferred-length character pointer of kind `C_CHAR`. It is an **INTENT (OUT)**
22 argument. FSTRPTR becomes **pointer associated** with the leftmost characters of the **actual**
23 **argument** element sequence (15.5.2.12) of CSTRARRAY if it appears, or with the leftmost char-
24 acters (in array element order) of the array S if CSTRPTR appears.

25 The length type parameter of FSTRPTR becomes the largest value for which no C null charac-
26 ters appear in the sequence, and which is less than or equal to NCHARS if present, and the size
27 of CSTRARRAY otherwise.

28 NCHARS (optional) shall be an integer scalar with a nonnegative value. It is an **INTENT (IN)** argument.
29 NCHARS shall be present if CSTRARRAY is **assumed-size**, or if CSTRPTR appears. If CSTRAR-
30 RAY appears, NCHARS shall not be greater than the size of CSTRARRAY.

31 4 If `C_CHAR` has the value `-1`, indicating that there is no C character kind, the generic subroutine `C_F_-`
32 `STRPOINTER` does not have any specific procedure.

33 5 **Example.**

34 *Case (i):* This interoperable procedure prints a C string to a Fortran file.

```

35      Subroutine logstring (str) Bind (C)
36      Use Iso_C_Binding
37      Character (Kind=C_char), Dimension(*), Target :: str

```

```

1      Character (:, C_char), Pointer :: sval
2      Integer, Parameter :: logunit = 17
3      Call C_F_Strpointer (str, sval, 1020) ! Max length for sval is 1020.
4      Write (logunit, *) 'C: ', sval
5  End Subroutine

```

Case (ii): This program shows how to use C_F_STRPTR to display the result of calling the C library function getenv.

```

6      Program cfs_example
7      Use Iso_C_Binding
8      Character (:, C_char), Pointer :: eval
9      Type (C_ptr) :: envptr
10     Interface
11     Function getenv (name) Bind (C)
12     Import C_char, C_ptr
13     Character (Kind=C_char), Intent (In) :: name (*)
14     Type (C_ptr) :: getenv
15     End Function
16 End Interface
17 envptr = getenv ("CFS"/C_Null_Char)
18 If (C_associated (envptr)) Then
19     Call C_F_Strpointer (envptr, eval, 1023) ! Max length 1023.
20     Print *, 'CFS value is ', eval, ''
21 Else
22     Print *, 'CFS has no value'
23 End If
24 End Program

```

18.2.3.6 C_FUNLOC (X)

1 **Description.** C address of the argument.

2 **Class.** Transformational function.

3 **Argument.** X shall be a procedure; if it is a procedure pointer it shall be associated. It shall not be a coindexed object.

4 **Result Characteristics.** Scalar of type C_FUNPTR.

5 **Result Value.** The result value is described using the result name FUNPTR. The result is determined as if C_FUNPTR were a derived type containing a procedure pointer component PX with an implicit interface and the pointer assignment FUNPTR%PX => X were executed. The result value can be used as an actual CPTR argument in a call to C_F_PROCPTR where the FPTR argument has attributes that would allow the pointer assignment FPTR => X. Such a call to C_F_PROCPTR shall have the effect of the pointer assignment FPTR => X.

6 **Example.** This code fragment shows how C_FUNLOC can be used to register an “atexit” procedure with the C library.

```

41     Use Iso_C_Binding
42     Interface
43     Function atexit (func) Bind (C)
44     Import
45     Integer (C_int) :: atexit
46     Type (C_funptr), Value :: func
47     End Function
48     Subroutine my_atexit_sub() Bind(C)
49     End Subroutine
50 End Interface

```

```

1      Integer (C_int) :: errno
2      errno = atexit (C_funloc (my_atexit_sub))
3      If (errno==0) Then
4          Print *, 'At exit sub registered'
5      Else
6          Print *, 'Error', errno, 'from atexit'
7      End If

```

18.2.3.7 C_LOC (X)

1 **Description.** [C address](#) of the argument.

2 **Class.** [Transformational function](#).

3 **Argument.** X shall have either the [POINTER](#) or [TARGET](#) attribute. It shall not be a [coindexed object](#). It shall be a variable with [interoperable](#) type and [kind type parameters](#), an [assumed-type](#) variable, or a non-polymorphic variable that has no [length type parameter](#). 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.

Case (i): 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.

Case (ii): 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.

Case (iii): 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 9899:2018, 6.5.3.2) to the [target](#) of CPTR%PX.

The result value 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.

6 **Example.** This function uses C_LOC to return the address of a Fortran floating-point vector to a C caller.

```

31      Function new_fortran_float_vec (n) Bind (C) Result (r)
32          Use Iso_C_Binding
33          Integer (C_size_t), Value :: n
34          Type (C_ptr) :: r
35          Real (C_float), Pointer :: rp (:)
36          Allocate (rp (n), Stat=istat)
37          If (istat==0) Then
38              r = C_loc (rp (1))
39          Else
40              r = C_null_ptr
41          End If
42      End Function

```

An example using C_LOC on an array of noninteroperable type appears in *Case (ii)* of the Examples paragraph of [18.2.3.3](#).

NOTE

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 (ISO/IEC 9899:2018, 6.5.3.2) is likely to be portable.

18.2.3.8 C_SIZEOF (X)

1 **Description.** Size of X in bytes.

2 **Class.** [Inquiry function](#).

3 **Argument.** X shall be a [data entity](#) with [interoperable](#) type and type parameters, and shall not be an
4 [assumed-size array](#), an [assumed-rank array](#) that is associated with an [assumed-size array](#), an unallocated
5 allocatable variable, or a pointer that is not associated.

6 **Result Characteristics.** Scalar integer of kind C_SIZE_T ([18.3.1](#)).

7 **Result Value.**

8 *Case (i):* If X is scalar, the result value is the value that the [companion processor](#) returns as the result
9 of applying the sizeof operator (ISO/IEC 9899:2018, 6.5.3.4) to an object of a type that inter-
10 operates with the type and type parameters of X.

11 *Case (ii):* If X is an array, the result value is the value that the [companion processor](#) returns as the result
12 of applying the sizeof operator to an object of a type that interoperates with the type and type
13 parameters of X, multiplied by the number of elements in X.

14 **Example.** With eight-bit bytes and the declaration INTEGER (C_INT32_T) :: X (3), the result value of C_-
15 SIZEOF (X) is twelve.

18.2.3.9 F_C_STRING (STRING [, ASIS])

1 **Description.** String with appended null character.

2 **Class.** [Transformational function](#).

3 **Arguments.**

4 STRING shall be a character scalar of kind [C_CHAR](#). If [C_CHAR](#) has the value -1 , indicating that there
5 is no C character kind, the generic function F_C_STRING has no specific procedure.

6 ASIS (optional) shall be a logical scalar.

7 **Result Characteristics.** Character scalar of kind [C_CHAR](#). If ASIS is present with the value true, the length
8 type parameter of the result is equal to one plus the length of STRING, otherwise it is equal to one plus the
9 length of STRING without trailing blanks.

10 **Result Value.** The leftmost characters of the result, up to the penultimate character, are equal to the cor-
11 responding characters of STRING. The final character of the result is equal to [C_NULL_CHAR](#).

12 **Example.** If X is declared as CHARACTER(6,C_CHAR), and has the value 'abc ' (with three trailing
13 blanks), then F_C_STRING (X, .TRUE.) has length seven and the value 'abc '//C_NULL_CHAR, and F_C_-
14 STRING (X) has length four and the value 'abc '//C_NULL_CHAR.

18.3 Interoperability between Fortran and C entities

18.3.1 Interoperability of intrinsic types

Table 18.2 shows the interoperability between Fortran intrinsic types and C types. A Fortran intrinsic type with particular type parameter values is **interoperable** with a C type if the type and **kind type parameter** value are listed in the table on the same row as that C type. If the type is character, the **length type parameter** is **interoperable** if and only if its value is one. A combination of Fortran type and type parameters that is **interoperable** with a C type listed in the table is also **interoperable** with any unqualified C type that is compatible with the listed C type.

The second column of the table refers to the **named constants** made accessible by the ISO_C_BINDING intrinsic module. If the value of any of these **named constants** is negative, there is no combination of Fortran type and type parameters **interoperable** with the C type shown in that row.

A combination of intrinsic type and type parameters is **interoperable** if it is **interoperable** with a C type. The C types mentioned in Table 18.2 are defined in ISO/IEC 9899:2018, 6.2.5, 7.19, and 7.20.1.

Table 18.2 — 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

Interoperability between Fortran and C types (cont.)

Fortran type	Named constant from the ISO_C_BINDING module (kind type parameter if value is positive)	C type
	C_LONG_DOUBLE_COMPLEX	long double _Complex
LOGICAL	C_BOOL	_Bool
CHARACTER	C_CHAR	char

NOTE

ISO/IEC 9899:2018 specifies that the representations for nonnegative signed integers are the same as the corresponding values of unsigned integers. Because Fortran does not provide direct support for unsigned kinds of integers, the ISO_C_BINDING module does not make accessible [named constants](#) for their [kind type parameter](#) values. A user can use the signed kinds of integers to interoperate with the unsigned types and all their qualified versions as well. This has the potentially surprising side effect that the C type unsigned char is [interoperable](#) with the type integer with a [kind type parameter](#) of C_SIGNED_CHAR.

18.3.2 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 1

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 9899:2018 does not require this to be the case.

NOTE 2

The function C_LOC can be used to return a value of type C_PTR that is the [C address](#) of an allocated [allocatable](#) variable. The function C_FUNLOC can be used to return a value of type C_FUNPTR that is the [C address](#) of a procedure. For C_LOC and C_FUNLOC the returned value is of an [interoperable](#) type and thus can be used in contexts where the procedure or [allocatable](#) variable is not directly allowed. For example, it could be passed as an [actual argument](#) to a C function.

Similarly, type C_FUNPTR or C_PTR can be used in a dummy argument or [structure component](#) and can have a value that is the [C address](#) of a procedure or [allocatable](#) variable, even in contexts where a procedure or [allocatable](#) variable is not directly allowed.

18.3.3 Interoperability of enum types

- 1 An enum type interoperates with its corresponding C enumerated type. It also interoperates with the C integer type that interoperates with its enumerators.

18.3.4 Interoperability of derived types and C structure types

- 1 Interoperability between a [derived type](#) in Fortran and a structure type in C is provided by the [BIND attribute](#) on the Fortran type.

C1801 (R726) A [derived type](#) with the [BIND attribute](#) shall not have the [SEQUENCE attribute](#).

C1802 (R726) A [derived type](#) with the [BIND attribute](#) shall not have [type parameters](#).

- 1 C1803 (R726) A **derived type** with the **BIND attribute** shall not have the **EXTENDS attribute**.
- 2 C1804 (R726) A **derived-type-def** that defines a **derived type** with the **BIND attribute** shall not have a **type-**
3 **bound-procedure-part**.
- 4 C1805 (R726) A **derived type** with the **BIND attribute** shall have at least one component.
- 5 C1806 (R726) Each component of a **derived type** with the **BIND attribute** shall be a nonpointer, nonalloc-
6 atable data component with **interoperable** type and **type parameters**.

NOTE 1

The syntax rules and their constraints require that a **derived type** that is **interoperable** with a C structure 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.

- 7 2 A **derived type** is **interoperable** with a C structure type if and only if the **derived type** has the **BIND attribute**
8 (7.5.2), the **derived type** and the C structure type have the same number of components, and the com-
9 ponents of the **derived type** would interoperate with corresponding components of the C structure type
10 as described in 18.3.5 and 18.3.6 if the components were variables. A component of a **derived type** and
11 a component of a C structure type correspond if they are declared in the same relative position in their
12 respective type definitions.

NOTE 2

The names of the corresponding components of the **derived type** and the C structure type need not be the same.

- 13 3 There is no Fortran type that is **interoperable** with a C structure type that contains a bit field or that contains
14 a flexible array member. There is no Fortran type that is **interoperable** with a C union type.

NOTE 3

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 structure type.

NOTE 4

ISO/IEC 9899:2018 requires the names and component names to be the same in order for the types to be compatible (ISO/IEC 9899:2018, 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 structure type because the case of identifiers is significant in C but not in Fortran.

18.3.5 Interoperability of scalar variables

- 1 A named scalar Fortran variable is **interoperable** if and only if its type and type parameters are **interoperable**, it is not a **coarray**, it has neither the **ALLOCATABLE** nor the **POINTER** attribute, and if it is of type character its length is not assumed or declared by an expression that is not a **constant expression**.
- 2 An **interoperable** scalar Fortran variable is **interoperable** with a scalar C entity if their types and type parameters are **interoperable**.

18.3.6 Interoperability of array variables

- 1 A Fortran variable that is a named array is **interoperable** if and only if its type and type parameters are **interoperable**, it is not a **coarray**, it is of explicit shape or assumed size, and if it is of type character its length is not assumed or declared by an expression that is not a **constant expression**.
- 2 An **explicit-shape** or **assumed-size** array of **rank** r , with a shape of $\begin{bmatrix} e_1 & \dots & e_r \end{bmatrix}$ 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 $\begin{bmatrix} e_1 & \dots & e_{r-1} \end{bmatrix}$, 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 1

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 2

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 3

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 4

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.

NOTE 4 (cont.)

Fortran's rules of sequence association (15.5.2.12) 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.

18.3.7, NOTE 4 has an example of interoperation between Fortran and C strings.

18.3.7 Interoperability of procedures and procedure interfaces

- 1 A Fortran procedure is **interoperable** if and only 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 9899:2018, 6.2.5, 7.19, and 7.20.1) of the formal parameter,
 - the dummy argument is a nonallocatable nonpointer variable of type CHARACTER with assumed character length and the formal parameter is a pointer to **CFI_cdesc_t**,
 - the dummy argument is **allocatable**, **assumed-shape**, **assumed-rank**, or a **pointer** without the **CONTIGUOUS attribute**, and the formal parameter is a pointer to **CFI_cdesc_t**, or
 - the dummy argument is **assumed-type** and not **allocatable**, **assumed-shape**, **assumed-rank**, or a **pointer**, and the formal parameter is a pointer to **void**,
 - (6) each **allocatable** or **pointer** dummy argument of type CHARACTER has **deferred character length**, and
 - (7) the prototype does not have variable arguments as denoted by the ellipsis (...).

NOTE 1

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 2

The C language allows specification of a C function that can take a variable number of arguments (ISO/IEC 9899:2018, 7.16). This document 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.

- 1 4 In a reference from C to a Fortran procedure with an interoperable [interface](#), a C actual argument shall be
 2 the address of a [C descriptor](#) for the intended [effective argument](#) if the corresponding [dummy argument](#)
 3 interoperates with a C formal parameter that is a pointer to [CFI_cdesc_t](#). In this [C descriptor](#), the mem-
 4 bers other than `attribute` and `type` shall describe an object with the same characteristics as the intended
 5 [effective argument](#). The value of the `attribute` member of the [C descriptor](#) shall be compatible with the
 6 characteristics of the [dummy argument](#). The `type` member shall have a value that depends on the intended
 7 [effective argument](#) as follows:
- 8 • if the [dynamic type](#) of the intended [effective argument](#) is an interoperable type listed in Table 18.4,
 9 the corresponding value for that type;
 - 10 • if the [dynamic type](#) of the intended [effective argument](#) is an [intrinsic type](#) for which the processor
 11 defines a nonnegative type specifier value not listed in Table 18.4, that type specifier value;
 - 12 • otherwise, `CFI_type_other`.
- 13 5 When an interoperable Fortran procedure that is invoked from C has a [dummy argument](#) with the [CON-](#)
 14 [TIGUOUS attribute](#) or that is an assumed-length CHARACTER [explicit-shape](#) or [assumed-size](#) array, and
 15 the actual argument is the address of a [C descriptor](#) for a discontinuous object, the Fortran processor shall
 16 handle the difference in contiguity.
- 17 6 When an interoperable C procedure whose Fortran interface has a [dummy argument](#) with the [CONTIGU-](#)
 18 [OUS attribute](#) or that is an assumed-length CHARACTER [explicit-shape](#) or [assumed-size](#) array is invoked
 19 from Fortran and the [effective argument](#) is discontinuous, the Fortran processor shall ensure that the C
 20 procedure receives a descriptor for a contiguous object.
- 21 7 If an interoperable procedure defined by means other than Fortran has an optional [dummy argument](#), and
 22 the corresponding [actual argument](#) in a reference from Fortran is absent, the procedure is invoked with
 23 a null pointer for that argument. If an interoperable procedure defined by means of Fortran is invoked
 24 by a C function, an optional [dummy argument](#) is absent if and only if the corresponding argument in the
 25 invocation is a null pointer.

NOTE 3

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 4

The interoperability of Fortran procedure [interfaces](#) with C function prototypes is only one part of invocation of a C function from Fortran. There are four pieces to consider in such an invocation: the procedure reference, the Fortran procedure [interface](#), the C function prototype, and the C function. Conversely, the invocation of a Fortran procedure from C involves the function reference, the C function prototype, the Fortran procedure [interface](#), and the Fortran procedure. In order to determine whether a reference is allowed, it is necessary to consider all four pieces.

For example, consider a C function that can be described by the C function prototype

```
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 ([15.5.2.12](#)). Those array [dummy arguments](#) in the procedure [interface](#) are [interoperable](#) with the formal parameters of the C function prototype. The C function is not shown here, but is assumed to be compatible with the C function prototype.

NOTE 5

If an interoperable C procedure whose Fortran interface has a [dummy argument](#) which has the [CONTIGUOUS attribute](#), or is an assumed-length CHARACTER [explicit-shape](#) or [assumed-size](#) array, is invoked from C, because the invoking routine is responsible for the contents of the C descriptor, it therefore might not describe a contiguous data object.

18.4 C descriptors

1

- 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.
- 2 This C structure is defined in the source file `ISO_Fortran_binding.h`.

3

18.5 The source file `ISO_Fortran_binding.h`

5

18.5.1 Summary of contents

6

- 1 The source file `ISO_Fortran_binding.h` shall contain the C structure definitions, typedef declarations, macro definitions, and function prototypes specified in [18.5.2](#) to [18.5.5](#). The definitions and declarations in `ISO_Fortran_binding.h` can be used by a C function to interpret and manipulate a [C descriptor](#). These provide a means to specify a C prototype that interoperates with a Fortran interface that has a noninteroperable dummy variable ([18.3.7](#)).

11

1 2 The source file `ISO_Fortran_binding.h` may be included in any order relative to the standard C headers,
 2 and may be included more than once in a given scope, with no effect different from being included only
 3 once, other than the effect on line numbers.

4 3 A C source file that includes the `ISO_Fortran_binding.h` header file shall not use any names starting with
 5 `CFI_` that are not defined in the header, and shall not define any of the structure names defined in the header
 6 as macro names. All names other than structure member names defined in the header begin with `CFI_` or
 7 an underscore character, or are defined by a standard C header that it includes.

8 18.5.2 The `CFI_dim_t` structure type

9 1 `CFI_dim_t` is a typedef name for a C structure. It is used to represent lower bound, extent, and memory
 10 stride information for one dimension of an array. The type `CFI_index_t` is described in 18.5.4. `CFI_dim_t`
 11 contains at least the following members in any order.

12 **`CFI_index_t lower_bound`;** The value is equal to the value of the lower bound for the dimension being
 13 described.

14 **`CFI_index_t extent`;** The value is equal to the number of elements in the dimension being described, or `-1`
 15 for the final dimension of an assumed-size array.

16 **`CFI_index_t sm`;** The value is equal to the memory stride for a dimension; this is the difference in bytes
 17 between the addresses of successive elements in the dimension being described.

18 18.5.3 The `CFI_cdesc_t` structure type

19 1 `CFI_cdesc_t` is a typedef name for a C structure, which contains a flexible array member. It shall contain at
 20 least the members described in this subclause. The values of these members of a structure of type `CFI-`
 21 `cdesc_t` that is produced by the functions and macros specified in this document, or received by a C function
 22 when invoked by a Fortran procedure, shall have the properties described in this subclause.

23 2 The first three members of the structure shall be `base_addr`, `elem_len`, and `version` in that order. The final
 24 member shall be `dim`. All other members shall be between `version` and `dim`, in any order. The types `CFI-`
 25 `attribute_t`, `CFI_rank_t`, and `CFI_type_t` are described in 18.5.4. The type `CFI_dim_t` is described in 18.5.2.

26 **`void * base_addr`;** If the object is an unallocated allocatable variable or a pointer that is disassociated, the
 27 value is a null pointer; otherwise, if the object has zero size, the value is not a null pointer but is oth-
 28 erwise processor-dependent. Otherwise, the value is the base address of the object being described.
 29 The base address of a scalar is its [C address](#). The base address of an array is the [C address](#) of the first
 30 element in Fortran array element order.

31 **`size_t elem_len`;** If the object is scalar, the value is the storage size in bytes of the object; otherwise, the
 32 value is the storage size in bytes of an element of the object.

33 **`int version`;** The value is equal to the value of `CFI_VERSION` in the source file `ISO_Fortran_binding.h` that
 34 defined the format and meaning of this [C descriptor](#).

35 **`CFI_rank_t rank`;** The value is equal to the number of dimensions of the Fortran object being described; if
 36 the object is scalar, the value is zero.

- 1 **CFI_type_t type;** The value is equal to the specifier for the type of the object. Each interoperable intrinsic
 2 C type has a specifier. Specifiers are also provided to indicate that the type of the object is an inter-
 3 operable structure, or is unknown. The macros listed in Table 18.4 provide values that correspond
 4 to each specifier.
- 5 **CFI_attribute_t attribute;** The value is equal to the value of an attribute code that indicates whether the
 6 object described is [allocatable](#), a [data pointer](#), or a nonallocatable nonpointer data object. The macros
 7 listed in Table 18.3 provide values that correspond to each code.
- 8 **CFI_dim_t dim;** The number of elements in the `dim` array is equal to the rank of the object. Each element
 9 of the array contains the lower bound, extent, and memory stride information for the corresponding
 10 dimension of the Fortran object.
- 11 3 For a [C descriptor](#) of an array pointer or [allocatable](#) array, the value of the `lower_bound` member of each
 12 element of the `dim` member of the descriptor is determined by [argument association](#), allocation, or [pointer](#)
 13 [association](#). For a [C descriptor](#) of a nonallocatable nonpointer object, the value of the `lower_bound` member
 14 of each element of the `dim` member of the descriptor is zero.
- 15 4 There shall be an ordering of the dimensions such that the absolute value of the `sm` member of the first
 16 dimension is not less than the `elem_len` member of the [C descriptor](#) and the absolute value of the `sm` mem-
 17 ber of each subsequent dimension is not less than the absolute value of the `sm` member of the previous
 18 dimension multiplied by the extent of the previous dimension.
- 19 5 In a [C descriptor](#) of an [assumed-size array](#), the `extent` member of the last element of the `dim` member has
 20 the value -1 .

NOTE 1

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 2

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.

21 18.5.4 Macros and typedefs in ISO_Fortran_binding.h

- 22 1 Except for CFI_CDESC_T, each macro defined in `ISO_Fortran_binding.h` expands to an integer constant
 23 expression that is either a single token or a parenthesized expression that is suitable for use in `#if` prepro-
 24 cessing directives.
- 25 2 CFI_CDESC_T is a function-like macro that takes one argument, which is the rank of the [C descriptor](#) to
 26 create, and evaluates to an unqualified type of suitable size and alignment for defining a variable to use
 27 as a [C descriptor](#) of that rank. The argument shall be an integer constant expression with a value that is
 28 greater than or equal to zero and less than or equal to CFI_MAX_RANK. A pointer to a variable declared
 29 using CFI_CDESC_T can be cast to `CFI_cdesc_t*`. A variable declared using CFI_CDESC_T shall not have an
 30 initializer.

NOTE 1

The CFI_CDESC_T macro provides the memory for a [C descriptor](#). The address of an entity declared using the macro is not usable as an actual argument corresponding to a formal parameter of type `CFI_cdesc_t *` without an explicit cast. For example, the following code uses CFI_CDESC_T to declare a [C descriptor](#) of rank 5 and pass it to [CFI_deallocate](#) (18.5.5.4).

```
CFI_CDESC_T(5) object;
int ind;
... Code to define and use C descriptor.
ind = CFI_deallocate((CFI_cdesc_t *)&object);
```

- 1 3 CFI_index_t is a typedef name for a standard signed integer type capable of representing the result of sub-
- 2 tracting two pointers.
- 3 4 The CFI_MAX_RANK macro has a processor-dependent value equal to the largest rank supported. The value
- 4 shall be greater than or equal to 15. CFI_rank_t is a typedef name for a standard integer type capable of
- 5 representing the largest supported rank.
- 6 5 The CFI_VERSION macro has a processor-dependent value that encodes the version of the `ISO_Fortran_`
- 7 `binding.h` source file containing this macro. This value should be increased if a new version of the source
- 8 file is incompatible with the previous version.
- 9 6 The macros in Table 18.3 are for use as attribute codes. The values shall be nonnegative and distinct. CFI_
- 10 attribute_t is a typedef name for a standard integer type capable of representing the values of the attribute
- 11 codes.

Table 18.3 — 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

- 12 7 CFI_attribute_pointer specifies a data object with the Fortran POINTER attribute. CFI_attribute_allocatable
- 13 specifies an object with the Fortran ALLOCATABLE attribute. CFI_attribute_other specifies a nonallocatable
- 14 nonpointer object.
- 15 8 The macros in Table 18.4 are for use as type specifiers. The value for CFI_type_other shall be negative and
- 16 distinct from all other type specifiers. CFI_type_struct specifies a C structure that is interoperable with a
- 17 Fortran derived type; its value shall be positive and distinct from all other type specifiers. If a C type is not
- 18 interoperable with a Fortran type and kind supported by the Fortran processor, its macro shall evaluate to
- 19 a negative value. Otherwise, the value for a macro listed in Table 18.4 shall be positive.
- 20 9 If the processor supports interoperability of a Fortran [intrinsic type](#) with a C type not listed in Table 18.4,
- 21 the processor shall define a type specifier value for that type which is positive and distinct from all other
- 22 type specifiers.
- 23 10 CFI_type_t is a typedef name for a standard integer type capable of representing the values for the suppor-
- 24 ted type specifiers.

Table 18.4 — 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
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 2

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.

- 1 11 The macros in Table 18.5 are for use as error codes. The macro CFI_SUCCESS shall be defined to be the
2 integer constant zero. The value of each macro other than CFI_SUCCESS shall be nonzero and shall be
3 different from the values of the other macros specified in this subclause. Error conditions other than those
4 listed in this subclause should be indicated by error codes different from the values of the macros named
5 in this subclause.
- 6 12 The values of the macros in Table 18.5 indicate the error condition described.

Table 18.5 — ISO_Fortran_binding.h macros for error codes

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	In a context that requires a null pointer value, the base address member of a C descriptor is not a null pointer.
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 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.

18.5.5 Functions declared in ISO_Fortran_binding.h

18.5.5.1 Arguments and results of the functions

Some of the functions described in [18.5.5](#) return an error indicator; this is an integer value that indicates whether an error condition was detected. The value zero indicates that no error condition was detected, and a nonzero value indicates which error condition was detected. Table [18.5](#) lists standard error conditions and macro names for their corresponding error codes. A processor is permitted to detect other error conditions. If an invocation of a function defined in [18.5.5](#) could detect more than one error condition and an error condition is detected, which error condition is detected is processor dependent.

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](#).

Prototypes for these functions, or equivalent macros, are provided in the `ISO_Fortran_binding.h` file as described in [18.5.5](#). It is unspecified whether the functions defined by this header are macros or identifiers declared with external linkage. If a macro definition is suppressed in order to access an actual function, the behavior is undefined.

NOTE

These functions are allowed to be macros to provide extra implementation flexibility. For example, `CFI_establish` could include the value of `CFI_VERSION` in the header used to compile the call to `CFI_establish` as an extra argument of the actual function used to establish the [C descriptor](#).

18.5.5.2 The CFI_address function

Synopsis. C address of an object described by a [C descriptor](#).

```
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](#) except for the last dimension of a [C descriptor](#) for an [assumed-size array](#). For the [C descriptor](#) of an [assumed-size array](#), the value of the subscript for the last dimension shall not be less than the lower bound, and the subscript order value specified by the `subscripts` shall not exceed the size of the array.

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 );
```

18.5.5.3 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, if `dv->rank`>0.

`upper_bounds` shall be the address of an array with at least `dv->rank` elements, if `dv->rank`>0.

`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

1 the type specified in the [C descriptor](#) is a character type, the supplied element length overrides the current
 2 element-length information in the descriptor.

3 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

```
6 REAL, ALLOCATABLE :: A(:, :)
```

7 and the array is not allocated, the following code allocates it to be of shape `[100, 500]`:

```
8 CFI_index_t lower[2], upper[2];
9 int ind;
10 lower[0] = 1; lower[1] = 1;
11 upper[0] = 100; upper[1] = 500;
12 ind = CFI_allocate(dv, lower, upper, 0);
```

13 18.5.5.4 The CFI_deallocate function

14 **1 Synopsis.** Deallocate memory for an object described by a [C descriptor](#).

```
15 int CFI_deallocate(CFI_cdesc_t *dv);
```

16 **2 Formal Parameter.** `dv` shall be the address of a [C descriptor](#) describing the object. It shall have been alloc-
 17 ated using the same mechanism as the Fortran [ALLOCATE statement](#). If the object is a pointer, it shall be
 18 associated with a target satisfying the conditions for successful deallocation by the Fortran [DEALLOCATE](#)
 19 [statement \(9.7.3\)](#).

20 **3 Description.** Successful execution of `CFI_deallocate` deallocates memory for the object using the same
 21 mechanism as the Fortran [DEALLOCATE statement](#), and the `base_addr` member of the [C descriptor](#) be-
 22 comes a null pointer.

23 If an error is detected, the [C descriptor](#) is not modified.

24 **4 Result Value.** The result is an error indicator.

25 **5 Example.** If `dv` is the address of a [C descriptor](#) for the Fortran array `A` declared as

```
26 REAL, ALLOCATABLE :: A(:, :)
```

27 and the array is allocated, the following code deallocates it:

```
28 int ind;
29 ind = CFI_deallocate(dv);
```

30 18.5.5.5 The CFI_establish function

31 **1 Synopsis.** Establish a [C descriptor](#).

```
32 int CFI_establish(CFI_cdesc_t *dv, void *base_addr, CFI_attribute_t attribute,
33                  CFI_type_t type, size_t elem_len, CFI_rank_t rank,
34                  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 **storage sequence** that is appropriately aligned (ISO/IEC 9899:2018, 3.2) for an object of the type specified by `type`.

`attribute` shall be one of the attribute codes in Table 18.3. If it is `CFI_attribute_allocatable`, `base_addr` shall be a null pointer. If it is `CFI_attribute_other`, `base_addr` shall not be a null pointer.

`type` shall have the value of one of the type codes in Table 18.4, or have a positive value corresponding to an interoperable C type.

`elem_len` If `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, `elem_len` will be ignored.

`rank` shall have a value in the range $0 \leq \text{rank} \leq \text{CFI\_MAX\_RANK}$. It specifies the rank of the object.

`extents` is ignored if `rank` is equal to zero or if `base_addr` is a null pointer. Otherwise, it shall be the address of an array with `rank` elements; the value of each element shall be nonnegative, and `extents[i]` specifies the extent of dimension *i* of the object.

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 or a disassociated pointer. 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 1

`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 2

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;
```

```

1      int ind;
2      rank = 1;
3      ind = CFI_establish((CFI_cdesc_t *)&field, NULL, CFI_attribute_allocatable,
4                          CFI_type_double, 0, rank, NULL);

```

Case (ii): Given the Fortran type definition

```

5      TYPE, BIND(C) :: T
6          REAL(C_DOUBLE) :: X
7          COMPLEX(C_DOUBLE_COMPLEX) :: Y
8      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:

```

13     typedef struct {double x; double _Complex y;} t;
14     t a_c[100];
15     CFI_CDESC_T(1) a_fortran;
16     int ind;
17     CFI_index_t extent[1];
18
19     extent[0] = 100;
20     ind = CFI_establish((CFI_cdesc_t *)&a_fortran, a_c, CFI_attribute_other,
21                         CFI_type_struct, sizeof(t), 1, extent);

```

18.5.5.6 The CFI_is_contiguous function

1 Synopsis. Test contiguity of an array.

```

24     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

Assumed-size and allocatable arrays are always contiguous, and therefore the result of `CFI_is_contiguous` on a C descriptor for such an array will be equal to 1.

18.5.5.7 The CFI_section function

1 Synopsis. Update a C descriptor for an array section for which each element is an element of a given array.

```

30     int CFI_section(CFI_cdesc_t *result, const CFI_cdesc_t *source,
31                    const CFI_index_t lower_bounds[], const CFI_index_t upper_bounds[],
32                    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`.

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 Description. Successful execution of `CFI_section` updates the `base_addr` and `dim` members of the **C descriptor** with the address `result` to describe the array section determined by `source`, `lower_bounds`, `upper_bounds`, and `strides`, as follows.

The array section is equivalent to the Fortran array section `SOURCE(sectsub1, sectsub2, ... sectsubn)`, where `SOURCE` is the array described by `source`, n is the rank of that array, and `sectsubi` is the subscript `loweri` if $stride_i$ is zero, and the section subscript `loweri : upperi : stridei` otherwise. The value of `loweri` is the lower bound of dimension i of `SOURCE` if `lower_bounds` is a null pointer and `lower_bounds[i]` otherwise. The value of `upperi` 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 `strides` is a null pointer and `strides[i]` otherwise. If $stride_i$ has the value zero, `loweri` shall have the same value as `upperi`.

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 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_bound;
upper[0] = source->dim[0].lower_bound + source->dim[0].extent - 1;
strides[0] = 1;
lower[1] = upper[1] = source->dim[1].lower_bound + 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);
```

18.5.5.8 The CFI_select_part function

- 1 Synopsis.** Update a [C descriptor](#) for an array section for which each element is a part of the corresponding element of an array.

```
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 an allocated [allocatable](#) array, an associated array [pointer](#), or a nonallocatable nonpointer array that is not [assumed-size](#).

displacement shall have a value $0 \leq \text{displacement} \leq \text{source->elem\_len} - 1$, and the sum of the displacement and the size in bytes of an element of the array section shall be less than or equal to `source->elem_len`. The address `displacement` bytes greater than the value of `source->base_addr` is the base of the array section and shall be appropriately aligned (ISO/IEC 9899:2018, 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, `elem_len` 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);
```

18.5.5.9 The CFI_setpointer function

- 1 Synopsis.** Update a [C descriptor](#) for a Fortran pointer to be associated with the whole of a given object or to be disassociated.

```
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 an allocated [allocatable](#) object, a [data pointer](#) object, or a nonallocatable nonpointer data object that is not an [assumed-size array](#). If **source** is not a null pointer, the corresponding values of the **rank** and **type** members shall be the same in the [C descriptors](#) with the addresses **source** and **result**. If **source** is not a null pointer and the C descriptor with the address **result** does not describe a deferred length character pointer, the corresponding values of the **elem_len** member 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 Description.** Successful execution of CFI_setpointer updates the **base_addr**, **dim**, and possibly **elem_len** 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 the [C descriptor](#) with the address **result** describes a character pointer with deferred length, the value of its **elem_len** member is set to **source->elem_len**.

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 **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);
```

18.6 Restrictions on C descriptors

- A [C descriptor](#) shall not be initialized, updated, or copied other than by calling the functions specified in [18.5.5](#).
- 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

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.

- 3 If the address of a **C descriptor** is a C actual argument that corresponds to an **assumed-shape** Fortran dummy argument, that descriptor shall not be for an **assumed-size array**.

18.7 Restrictions on formal parameters

- 1 Within a C function, an **allocatable** object shall be allocated or deallocated only by execution of the **CFI_allocate** and **CFI_deallocate** functions. A Fortran pointer can become associated with a target by execution of the **CFI_allocate** function.
- 2 Calling **CFI_allocate** or **CFI_deallocate** for a **C descriptor** changes the allocation status of the Fortran variable it describes.
- 3 If the address of an object is the value of a formal parameter that corresponds to a nonpointer dummy argument in an interface with the **BIND attribute**, then
 - if the dummy argument has the **INTENT (IN) attribute**, the object shall not be defined or become undefined, and
 - if the dummy argument has the **INTENT (OUT) attribute**, the object shall not be referenced before it is defined.
- 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**.

18.8 Restrictions on lifetimes

- 1 A **C descriptor** of, or C pointer to, any part of a Fortran object becomes undefined under the same conditions that the association status of a Fortran pointer **associated** with that object would become undefined, and any further use of it is undefined behavior (ISO/IEC 9899:2018, 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 9899:2018, 6.2.4) of the **C descriptor** shall not end before the return from the procedure call. If an object is

- 1 passed to a Fortran procedure as a nonallocatable, nonpointer dummy argument, its lifetime shall not end
2 before the return from the procedure call.
- 3 4 If the lifetime of a **C descriptor** for an allocatable object that was established by C ends before the program
4 exits, the object shall be unallocated at that time.
- 5 5 If a Fortran pointer becomes associated with a data object defined by the companion processor, the asso-
6 ciation status of the Fortran pointer becomes undefined when the lifetime of that data object ends.

NOTE

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.

7 18.9 Interoperation with C global variables

8 18.9.1 General

- 9 1 A C variable whose name has external linkage may interoperate with a **common block**¹ or with a variable
10 declared in the scope of a module. The **common block** or variable shall be specified to have the **BIND at-**
11 **tribute**.
- 12 2 At most one variable that is associated with a particular C variable whose name has external linkage is
13 permitted to be declared within all the Fortran **program units** of a program. A variable shall not be initially
14 defined by more than one processor.
- 15 3 If a **common block** is specified in a **BIND statement**, it shall be specified in a **BIND statement** with the same
16 **binding label** in each **scoping unit** in which it is declared. A C variable whose name has external linkage
17 interoperates with a **common block** that has been specified in a **BIND statement** if
- 18 • the C variable is of a structure type and the variables that are members of the **common block** are
19 **interoperable** with corresponding components of the structure type, or
 - 20 • the **common block** contains a single variable, and the variable is **interoperable** with the C variable.
- 21 4 There does not have to be an associated C entity for a Fortran entity with the **BIND attribute**.

NOTE

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.

¹Common blocks are obsolescent.

NOTE (cont.)

```

MODULE LINK_TO_C_VARS
  USE, INTRINSIC :: ISO_C_BINDING
  INTEGER(C_INT), BIND(C) :: C_EXTERN
  INTEGER(C_LONG) :: C2
  BIND(C, NAME='myVariable') :: C2

  COMMON /COM/ R, S
  REAL(C_FLOAT) :: R, S, T
  BIND(C) :: /COM/, /SINGLE/
  COMMON /SINGLE/ T
END MODULE LINK_TO_C_VARS

/* Global variables. */
int c_extern;
long myVariable;
struct { float r, s; } com;
float single;

```

18.9.2 Binding labels for common blocks and variables

- 1 The [binding label](#) of a variable or [common block](#)² is a default character value that specifies the name by which the variable or [common block](#) is known to the [companion processor](#).
- 2 If a variable or [common block](#) has the [BIND attribute](#) with the NAME= specifier and the value of its expression, after discarding leading and trailing blanks, has nonzero length, the variable or [common block](#) has this as its [binding label](#). The case of letters in the [binding label](#) is significant. If a variable or [common block](#) has the [BIND attribute](#) specified without a NAME= specifier, the [binding label](#) is the same as the name of the entity using lower case letters. Otherwise, the variable or [common block](#) has no [binding label](#).
- 3 The [binding label](#) of a C variable whose name has external linkage is the same as the name of the C variable. A Fortran variable or [common block](#) with the [BIND attribute](#) that has the same [binding label](#) as a C variable whose name has external linkage is [linkage associated](#) (19.5.1.5) with that variable.

18.10 Interoperation with C functions**18.10.1 Definition and reference of interoperable procedures**

- 1 A procedure that is [interoperable](#) may be defined either by means other than Fortran or by means of a Fortran subprogram, but not both. A C function that has an inline definition and no external definition is not considered to be defined in this sense.
- 2 If the procedure is defined by means other than Fortran,
 - it shall be describable by a C prototype that is [interoperable](#) with the [interface](#), and
 - if it is accessed using its [binding label](#), it shall
 - have a name that has external linkage as defined by ISO/IEC 9899:2018, 6.2.2, 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 9899:2018.

²Common blocks are obsolescent.

- 1 4 A reference in C to a procedure that has the **BIND attribute**, has the same **binding label**, and is defined by
 2 means of Fortran, causes the Fortran procedure to be invoked. A C function shall not invoke a function
 3 pointer whose value is the result of a reference to **C_FUNLOC** with a noninteroperable argument.
- 4 5 A procedure defined by means of Fortran shall not invoke setjmp or longjmp (ISO/IEC 9899:2018, 7.13).
 5 If a procedure defined by means other than Fortran invokes setjmp or longjmp, that procedure shall not
 6 cause any procedure defined by means of Fortran to be invoked. A procedure defined by means of Fortran
 7 shall not be invoked as a signal handler (ISO/IEC 9899:2018, 7.14.1).
- 8 6 If a procedure defined by means of Fortran and a procedure defined by means other than Fortran perform
 9 input/output operations on the same **external file**, the results are processor dependent (**12.5.4**).
- 10 7 If the value of a C function pointer will be the result of a reference to **C_FUNLOC** with a noninteroperable
 11 argument, it is recommended that the C function pointer be declared to have the type `void (*)()`.

12 18.10.2 Binding labels for procedures

- 13 1 The **binding label** of a procedure is a default character value that specifies the name by which a procedure
 14 with the **BIND attribute** is known to the **companion processor**.
- 15 2 If a procedure has the **BIND attribute** with the NAME= specifier and the value of its expression, after dis-
 16 carding leading and trailing blanks, has nonzero length, the procedure has this as its **binding label**. The case
 17 of letters in the **binding label** is significant. If a procedure has the **BIND attribute** with no NAME= specifier,
 18 and the procedure is not a **dummy procedure**, **internal procedure**, or **procedure pointer**, then the **binding**
 19 **label** of the procedure is the same as the name of the procedure using lower case letters. Otherwise, the
 20 procedure has no **binding label**.
- 21 C1807 A procedure defined in a submodule shall not have a **binding label** unless its **interface** is declared
 22 in the ancestor module.
- 23 3 The **binding label** for a C function whose name has external linkage is the same as the C function name.

NOTE

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

INTEGER(C_INT) FUNCTION C_FUNC() BIND(C, NAME="C_func")
  USE, INTRINSIC :: ISO_C_BINDING
...
END FUNCTION C_FUNC
```

ISO/IEC 9899:2018 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**.

24 18.10.3 Exceptions and IEEE arithmetic procedures

- 25 1 A procedure defined by means other than Fortran shall not use signal (ISO/IEC 9899:2018, 7.14.1) to
 26 change the **handling of any exception** that is being handled by the Fortran processor.

1 2 A procedure defined by means other than Fortran shall not alter the floating-point status (17.7) other than
2 by setting an exception flag to signaling.

3 3 The values of the floating-point exception flags on entry to a procedure defined by means other than Fortran
4 are processor dependent.

5 18.10.4 Asynchronous communication

6 1 Asynchronous communication for a Fortran variable with the [ASYNCHRONOUS attribute](#) occurs through
7 the action of procedures defined by means other than Fortran. It is initiated by execution of an asynchron-
8 ous communication initiation procedure and completed by execution of an asynchronous communication
9 completion procedure. Between the execution of the initiation and completion procedures, any variable
10 of which any part is associated with any part of the asynchronous communication variable is a pending
11 communication affector. Whether a procedure is an asynchronous communication initiation or comple-
12 tion procedure is processor dependent.

13 2 Asynchronous communication is either input communication or output communication. For input commu-
14 nication, a pending communication affector shall not be referenced, become defined, become undefined,
15 become associated with a dummy argument that has the [VALUE attribute](#), or have its [pointer association](#)
16 status changed. For output communication, a pending communication affector shall not be redefined, be-
17 come undefined, or have its [pointer association](#) status changed. The restrictions for asynchronous input
18 communication are the same as for asynchronous input data transfer. The restrictions for asynchronous
19 output communication are the same as for asynchronous output data transfer.

NOTE

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
... Code that involves BUF
```

In this example, there is asynchronous input communication and BUF is a pending communication affector between the two calls. MPI_IRecv can return while the communication (reading values into BUF) is still underway. The intent is that the code between MPI_IRecv and MPI_WAIT can execute without waiting for this communication to complete.

Similar code with the call of MPI_IRecv replaced by a call of MPI_Isend is asynchronous output communication.

19 Scope, association, and definition

19.1 Scopes, identifiers, and entities

1 An entity is identified by an identifier.

2 The scope of

- a global identifier is a program (5.2.2),
- a local identifier is an [inclusive scope](#),
- an identifier of a [construct entity](#) is that construct (10.2.4, 11.1), and
- an identifier of a [statement entity](#) is that statement or part of that statement (6.3),

excluding any nested scope where the identifier is treated as the identifier of a different entity (19.3, 19.4), or where an [IMPORT statement](#) (8.8) makes the identifier inaccessible.

3 An entity may be identified by

- an [image index](#) (3.81),
- a [name](#) (3.100),
- a [statement label](#) (3.132),
- an [external input/output unit](#) number (12.5),
- an identifier of a pending data transfer operation (12.6.2.9, 12.7),
- a submodule identifier (14.2.3),
- a [generic identifier](#) (3.75), or
- a [binding label](#) (3.15).

4 By means of association, an entity may be referred to by the same identifier or a different identifier in a different scope, or by a different identifier in the same scope.

19.2 Global identifiers

1 [Program units](#), [common blocks](#)¹, [external procedures](#), entities with [binding labels](#), [external input/output units](#), pending data transfer operations, and [images](#) are global entities of a program. The name of a [common block](#) with no [binding label](#), [external procedure](#) with no [binding label](#), or [program unit](#) that is not a submodule is a global identifier. The submodule identifier of a submodule is a global identifier. A [binding label](#) of an entity of the program is a global identifier. An entity of the program shall not be identified by more than one [binding label](#).

2 The global identifier of an entity shall not be the same as the global identifier of any other entity. Furthermore, a [binding label](#) shall not be the same as the global identifier of any other global entity, ignoring differences in case. A processor may assign a global identifier to an entity that is not specified by this document to have a global identifier (such as an intrinsic procedure); in such a case, the processor shall ensure that this assigned global identifier differs from all other global identifiers in the program.

¹Common blocks are obsolescent.

NOTE 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 2

Submodule identifiers are global identifiers, but because they consist of a module name and a [descendant](#) submodule name, the name of a submodule can be the same as the name of another submodule so long as they do not have the same ancestor module.

19.3 Local identifiers

19.3.1 Classes of local identifiers

1 Identifiers of entities, other than [statement](#) or [construct](#) entities ([19.4](#)), in the classes

(1) named variables, [named constants](#), named [procedure pointers](#), named constructs, statement functions², [internal procedures](#), [module procedures](#), [dummy procedures](#), [intrinsic procedures](#), [external procedures](#) that have [binding labels](#), intrinsic modules, [abstract interfaces](#), [generic interfaces](#), nonintrinsic 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 blocks³ that have [binding labels](#)

are local identifiers.

2 Within its scope, a local identifier of an entity of class (1) or class (4) shall not be the same as a global identifier used in that scope unless the global identifier

- is used only as the *use-name* of a [rename](#) in a [USE statement](#),
- is a [common block](#) name ([19.3.2](#)) and the local identifier is not that of a [named constant](#) or [intrinsic procedure](#),
- is an [external procedure](#) name that is also a generic name, or
- is an external function name and the [inclusive scope](#) is its defining subprogram ([19.3.3](#)).

3 Within its scope, a local identifier of one class shall not be the same as another local identifier of the same class, except that a generic name may be the same as the name of a procedure as explained in [15.4.3.4](#) or the same as the name of a derived type ([7.5.10](#)). A local identifier of one class may be the same as a local identifier of another class.

NOTE

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)
...
```

²Statement functions are obsolescent.

³Common blocks are obsolescent.

NOTE (cont.)

```

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*⁴ (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*⁵ has limited use within the scope of the subprogram in which the *entry-stmt* appears. It can be used to identify the *function result* if the subprogram is a function, to identify recursive references, or to identify a *common block* (the latter is possible only if the *entry-stmt* is in a module subprogram).

19.3.2 Local identifiers that are the same as common block names

- 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.

19.3.3 Function results

- 1 For each *FUNCTION statement* or *ENTRY statement* in a function subprogram, there is a *function result*. A *function result* is either a variable or a *procedure pointer*, and thus the name of a *function result* is a local identifier of class (1).

19.3.4 Components, type parameters, and bindings

- 1 A component name has the scope of its derived-type definition. Outside the type definition, it can 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 can 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*.

⁴Common blocks are obsolescent.

⁵The *ENTRY statement* is obsolescent.

- 1 3 The **binding name** (7.5.5) of a **type-bound procedure** has the scope of its derived-type definition. Outside
2 of the derived-type definition, it can also appear as the *binding-name* in a procedure reference.
- 3 4 A generic **binding** for which the *generic-spec* is not a *generic-name* has a scope that consists of all **scoping**
4 **units** in which an entity of the type is accessible.
- 5 5 A component name or **binding name** can appear only in a scope in which it is accessible.
- 6 6 The accessibility of components and **bindings** is specified in 7.5.4.8 and 7.5.5.

19.3.5 Argument keywords

- 8 1 As an **argument keyword**, a **dummy argument** name in an **internal procedure**, **module procedure**, or an
9 **interface body** has a scope of the **scoping unit** of the **host** of the procedure or **interface body**. As an **argument**
10 **keyword**, the name of a **dummy argument** of a procedure declared by a **procedure declaration statement**
11 that specifies an **explicit interface** has a scope of the **scoping unit** containing the **procedure declaration**
12 **statement**. It may appear only in a procedure reference for the procedure of which it is a **dummy argument**.
13 If the procedure is accessible in another **scoping unit** by **use** or **host** association (19.5.1.3, 19.5.1.4), the
14 **argument keyword** is accessible for procedure references for that procedure in that **scoping unit**.
- 15 2 A **dummy argument** name in an **intrinsic** procedure has a scope as an **argument keyword** of the **scoping unit**
16 in which the reference to the procedure occurs. As an **argument keyword**, it may appear only in a procedure
17 reference for the procedure of which it is a **dummy argument**.

19.4 Statement and construct entities

- 19 1 A variable that appears as a *data-i-do-variable* in a **DATA statement** or an *ac-do-variable* in an array con-
20 structor, as a dummy argument in a **statement function statement**⁶, or as an *index-name* in a **FORALL state-**
21 **ment**⁷ is a **statement entity**. Even if the name of a statement entity is the same as another identifier and the
22 statement is in the scope of that identifier, within the scope of the statement entity the name is interpreted
23 as that of the statement entity.
- 24 2 The name of a statement entity shall not be the same as an accessible global identifier or local identifier
25 of class (1) (19.3.1), except for a **common block**⁸ name or a scalar variable name. Within the scope of a
26 **statement entity**, another **statement entity** shall not have the same name.
- 27 3 A variable that appears as an *index-name* in a **FORALL**⁹ or **DO CONCURRENT** construct, as an *associate-*
28 *name* in an **ASSOCIATE**, **SELECT RANK**, **SELECT TYPE** construct, or as a *coarray-name* in a *codimension-*
29 *decl* in a **CHANGE TEAM** construct is a **construct entity**. A variable that has **LOCAL** or **LOCAL_INIT** locality
30 in a **DO CONCURRENT** construct is a **construct entity**. An entity that is declared in a specification in a
31 **BLOCK** construct, other than only in **ASYNCHRONOUS**, **IMPORT**, and **VOLATILE** statements, is a **construct**
32 **entity**. A **USE statement** in a **BLOCK** construct specifies that all the entities it accesses by **use association**
33 are construct entities. If an entity is a construct entity instead of a host entity only because it is wholly or
34 partially initialized in a **DATA statement**, the construct entity shall not be used prior to the **DATA statement**.
- 35 4 Two **construct entities** of the same construct shall not have the same identifier.

⁶Statement functions are obsolescent.

⁷The **FORALL statement** is obsolescent.

⁸Common blocks are obsolescent.

⁹The **FORALL construct** is obsolescent.

- 1 5 The name of a *data-i-do-variable* in a **DATA statement** or an *ac-do-variable* in an array constructor has a
 2 scope of its *data-implied-do* or *ac-implied-do*. It is a scalar variable. If *integer-type-spec* appears in *data-*
 3 *implied-do* or *ac-implied-do-control* it has the specified type and type parameters; otherwise it has the type
 4 and type parameters that it would have if it were the name of a variable in the innermost executable con-
 5 struct or **scoping unit** that includes the **DATA statement** or array constructor, and this type shall be integer
 6 type. It has no other attributes. The appearance of a name as a *data-i-do-variable* of an implied DO in a
 7 **DATA statement** or an *ac-do-variable* in an array constructor is not an implicit declaration of a variable
 8 whose scope is the **scoping unit** that contains the statement.
- 9 6 The name of a variable that appears as an *index-name* in a **DO CONCURRENT construct**, **FORALL statement**,
 10 or **FORALL construct** has a scope of the statement or construct. It is a scalar variable. If *integer-type-spec*
 11 appears in *concurrent-header* it has the specified type and type parameters; otherwise it has the type and
 12 type parameters that it would have if it were the name of a variable in the innermost executable construct
 13 or **scoping unit** that includes the **DO CONCURRENT** or **FORALL**, and this type shall be integer type. It has
 14 no other attributes. The appearance of a name as an *index-name* in a **DO CONCURRENT construct**, **FORALL**
 15 **statement**, or **FORALL construct** is not an implicit declaration of a variable whose scope is the **scoping unit**
 16 that contains the statement or construct.
- 17 7 A variable that has **LOCAL** or **LOCAL_INIT** locality in a **DO CONCURRENT construct** has the scope of that
 18 construct. Its attributes are specified in 11.1.7.5.
- 19 8 If *integer-type-spec* does not appear in a *concurrent-header*, an *index-name* shall not be the same as an ac-
 20 cessible global identifier, local identifier, or identifier of an outer construct entity, except for a **common**
 21 **block** name or a scalar variable name. An *index-name* of a contained **DO CONCURRENT construct**, **FOR-**
 22 **ALL statement**, or **FORALL construct** shall not be the same as an *index-name* of any of its containing **DO**
 23 **CONCURRENT** or **FORALL** constructs.
- 24 9 The **associate names** of an **ASSOCIATE construct** have the scope of the block. They have the **declared type**,
 25 **dynamic type**, **type parameters**, **rank**, and **bounds** specified in 11.1.3.2.
- 26 10 The **associate names** of a **CHANGE TEAM construct** have the scope of the block. They have the **declared**
 27 **type**, **dynamic type**, **type parameters**, **rank**, **corank**, **bounds**, and **cobounds** specified in 11.1.5.
- 28 11 The **associate name** of a **SELECT RANK construct** has a separate scope for each block of the construct. It
 29 has the attributes specified in 11.1.10.3.
- 30 12 The **associate name** of a **SELECT TYPE construct** has a separate scope for each block of the construct. Within
 31 each block, it has the **declared type**, **dynamic type**, **type parameters**, **rank**, and **bounds** specified in 11.1.11.2.
- 32 13 The name of a variable that appears as a dummy argument in a **statement function statement**¹⁰ has a scope
 33 of the statement in which it appears. It is a scalar that has the type and type parameters that it would have
 34 if it were the name of a variable in the **scoping unit** that includes the **statement function**; it has no other
 35 attributes.

¹⁰Statement functions are obsolescent.

19.5 Association

19.5.1 Name association

19.5.1.1 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.

19.5.1.2 Argument association

- 1 The rules governing **argument association** are given in Clause 15. As explained in 15.5, execution of a procedure reference establishes a correspondence between each **actual argument** and a **dummy argument** and thus an association between each present **dummy argument** and its **effective argument**. Argument association can be sequence association (15.5.2.12).
- 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

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 can be associated with an entirely different **effective argument** in a subsequent invocation of the procedure.

19.5.1.3 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 14.2.2. They allow for renaming of entities being accessed. **Use association** allows access in one scope to entities defined or declared in another scope; it remains in effect throughout the execution of the program.

19.5.1.4 Host association

- 1 A **derived-type definition**, **interface body**, **internal subprogram**, **module subprogram**, or **submodule** has access to entities from its host as specified in 8.8. A host-associated variable is considered to have been previously declared; any other host-associated entity is considered to have been previously defined. 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, nonintrinsic 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**

(8.5.9) within the **scoping unit**, or a name that appears within the **scoping unit** as a *module-name* in a *use-stmt* is a global identifier; any entity of the **host** that has this as its nongeneric name is inaccessible by that name. A name that appears in the **scoping unit** as

- (1) a *function-name* in a *stmt-function-stmt*¹¹ 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*¹²,
- (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*¹³,
- (11) a *dummy-arg-name* in a *function-stmt*, in a *subroutine-stmt*, in an *entry-stmt*¹⁴, or in a *stmt-function-stmt*¹⁵,
- (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) an *enum-type-name* in an *enum-def*,
- (17) an *enumeration-type-name* in an *enumeration-type-stmt*,
- (18) a *generic-name* in a *generic-spec* in an *interface-stmt*, or
- (19) 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 1

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 defined¹⁶ prior to the **DATA statement**.

¹¹Statement functions are obsolescent.

¹²Common blocks are obsolescent.

¹³The **EQUIVALENCE statement** is obsolescent.

¹⁴The **ENTRY statement** is obsolescent.

¹⁵Statement functions are obsolescent.

¹⁶Appearance of a **DATA statement** in the *execution-part* is obsolescent.

- 1 4 If a derived-type name of a **host** is inaccessible, data entities of that type or subobjects of such data entities
2 still can be accessible.

NOTE 2

An **interface body** that is not a module procedure interface body accesses by **host association** only those entities made accessible by **IMPORT statements**.

- 3 5 If an **external** or **dummy** procedure with an **implicit interface** is accessed via **host association**, then it shall
4 have the **EXTERNAL attribute** in the **host scoping unit**; if it is invoked as a function in the inner **scoping unit**,
5 its type and type parameters shall be established in the **host scoping unit**. The type and type parameters
6 of a function with the **EXTERNAL attribute** are established in a **scoping unit** if that **scoping unit** explicitly
7 declares them, invokes the function, accesses the function from a module, or accesses the function from its
8 **host** where its type and type parameters are established.
- 9 6 If an intrinsic procedure is accessed via **host association**, then it shall be established to be intrinsic in the
10 **host scoping unit**. An intrinsic procedure is established to be intrinsic in a **scoping unit** if that **scoping unit**
11 explicitly gives it the **INTRINSIC attribute**, invokes it as an intrinsic procedure, accesses it from a module,
12 or accesses it from its **host** where it is established to be intrinsic.

NOTE 3

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, but prevents CX from being declared locally.
    USE B, ONLY: BX ! Entities accessible are BX, and also IX and JX if
                   ! no other IX or JX is accessible to INNER_PROC.
                   ! Q is local to INNER_PROC, because it is a dummy argument.
    USE D, X => DX  ! Entities accessible are DX, DY, and DZ
                   ! X is local name for DX in INNER_PROC; if no other DX is
                   ! accessible in INNER_PROC, X and DX denote the same entity
    USE E, ONLY: EX ! EX is accessible in INNER_PROC, not in program A.
                   ! EY and EZ are not accessible in INNER_PROC or program A.
    USE G          ! FX and GX are accessible in INNER_PROC.
    ...
  END SUBROUTINE INNER_PROC
END PROGRAM A

```

Because program A contains the statement

```
USE B
```

all of the entities in module B, except for Q, are accessible in INNER_PROC, even though INNER_PROC contains the statement

```
USE B, ONLY: BX
```

The **USE statement** with the **ONLY** option means that this particular statement brings in only the entity named, not that this is the only variable from the module accessible in this **scoping unit**.

NOTE 4

For more examples of [host association](#), see [C.14.2](#).

19.5.1.5 Linkage association

Linkage association occurs between a module variable that has the [BIND attribute](#) and the C variable with which it interoperates, or between a Fortran [common block](#)¹⁷ and the C variable with which it interoperates ([18.9](#)). Such association remains in effect throughout the execution of the program.

19.5.1.6 Construct association

Execution of a [SELECT RANK](#) or [SELECT TYPE](#) statement establishes an association between the selector and the [associate name](#) of the construct. Execution of an [ASSOCIATE](#) or [CHANGE TEAM](#) statement establishes an association between each selector and the corresponding [associate name](#) of the construct.

In an [ASSOCIATE](#) or [SELECT TYPE](#) construct, the following rules apply.

- If a selector is [allocatable](#), it shall be allocated; the [associate name](#) is associated with the data object and does not have the [ALLOCATABLE attribute](#).
- If a selector has the [POINTER attribute](#), it shall be associated; the [associate name](#) is associated with the [target](#) of the pointer and does not have the [POINTER attribute](#).

If the selector is a variable other than an [array section](#) having a [vector subscript](#), the association is with the data object specified by the selector; otherwise, the association is with the value of the selector expression, which is evaluated prior to execution of the block.

Each [associate name](#) remains associated with the corresponding selector throughout the execution of the executed block. Within the block, each selector is known by and may be accessed by the corresponding [associate name](#). On completion of execution of the construct, the association is terminated.

NOTE

The association between the [associate name](#) and a data object is established prior to execution of the block and is not affected by subsequent changes to variables that were used in subscripts or substring ranges in the [selector](#).

19.5.2 Pointer association**19.5.2.1 General**

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](#). The definition status of an associated [data pointer](#) is that of its [target](#). If the pointer has deferred [type parameters](#) or shape, their values are assumed from the [target](#). If the pointer is [polymorphic](#), its [dynamic type](#) is assumed from the [dynamic type](#) of the [target](#).

19.5.2.2 Pointer association status

A pointer has 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

¹⁷Common blocks are obsolescent.

an initial association status of **undefined**. A pointer may be initialized to have an association status of **disassociated** or associated.

NOTE

A pointer from a module **program unit** might be accessible in a subprogram via **use association**. Such pointers have a lifetime that is greater than **targets** that are declared in the subprogram, unless such **targets** are **saved**. Therefore, if such a pointer is associated with a local **target**, there is the possibility that when a procedure defined by the subprogram completes execution, the **target** will cease to exist, leaving the pointer “dangling”. This document considers such pointers to have an undefined association status. They are neither associated nor **disassociated**. They cannot be used again in the program until their status has been reestablished. A processor is not required to detect when a pointer **target** ceases to exist.

19.5.2.3 Events that cause pointers to become associated

1 A pointer becomes associated when any of the following events occur.

- (1) The pointer is allocated (9.7.1) as the result of the successful execution of an **ALLOCATE statement** referencing the pointer.
- (2) The pointer is pointer-assigned to a **target** (10.2.2) that is associated or is specified with the **TARGET attribute** and, if **allocatable**, is allocated.
- (3) The pointer is a **subobject** of an object that is allocated by an **ALLOCATE statement** in which **SOURCE=** appears and the corresponding **subobject** of *source-expr* is associated.
- (4) The pointer is a dummy argument and its corresponding **actual argument** is not a pointer.
- (5) The pointer is a **default-initialized subcomponent** of an object, the corresponding initializer is not a reference to the intrinsic function **NULL**, and
 - (a) a procedure is invoked with this object as an **actual argument** corresponding to a non-pointer nonallocatable dummy argument with **INTENT (OUT)**,
 - (b) a procedure with this object as an **unsaved** nonpointer nonallocatable local variable is invoked,
 - (c) a **BLOCK construct** is entered and this object is an **unsaved** local nonpointer nonallocatable local variable of the **BLOCK construct**, or
 - (d) this object is allocated other than by an **ALLOCATE statement** in which **SOURCE=** appears.

19.5.2.4 Events that cause pointers to become disassociated

1 A pointer becomes **disassociated** when

- (1) the pointer is nullified (9.7.2),
- (2) the pointer is deallocated (9.7.3),
- (3) the pointer is **pointer-assigned** (10.2.2) to a **disassociated** pointer,
- (4) the pointer is a **subobject** of an object that is allocated by an **ALLOCATE statement** in which **SOURCE=** appears and the corresponding **subobject** of *source-expr* is **disassociated**, or
- (5) the pointer is a **default-initialized subcomponent** of an object, the corresponding initializer is a reference to the intrinsic function **NULL**, and
 - (a) a procedure is invoked with this object as an **actual argument** corresponding to a non-pointer nonallocatable dummy argument with **INTENT (OUT)**,
 - (b) a procedure with this object as an **unsaved** nonpointer nonallocatable local variable is invoked,

- (c) a **BLOCK construct** is entered and this object is an **unsaved** local nonpointer nonallocatable local variable of the **BLOCK construct**, or
- (d) this object is allocated other than by an **ALLOCATE statement** in which **SOURCE=** appears.

19.5.2.5 Events that cause the association status of pointers to become undefined

- 1 The association status of a pointer becomes undefined when
 - (1) the pointer is pointer-assigned to a **target** that has an undefined association status,
 - (2) the pointer is pointer-assigned to a **target** on a different **image**,
 - (3) the **target** of the pointer is deallocated other than through the pointer,
 - (4) the **target** of the pointer is a data object defined by the companion processor and the lifetime of that data object ends,
 - (5) the allocation transfer procedure (16.9.147) is executed, the pointer is associated with the argument FROM, and the argument TO does not have the **TARGET attribute**,
 - (6) completion of execution of an instance of a subprogram causes the pointer's **target** to become undefined (item (3) of 19.6.6),
 - (7) completion of execution of a **BLOCK construct** causes the pointer's **target** to become undefined (item (23) of 19.6.6),
 - (8) execution of the **host instance** of a **procedure pointer** is completed,
 - (9) execution of an instance of a subprogram completes and the pointer is declared or accessed in the subprogram that defines the procedure if the pointer
 - (a) does not have the **SAVE attribute**,
 - (b) is not in **blank common**¹⁸,
 - (c) is not in a named **common block** that is declared in any other **scoping unit** that is in execution,
 - (d) is not accessed by **host association**, and
 - (e) is not the result of a function declared to have the **POINTER attribute**,
 - (10) execution of an instance of a subprogram completes, the pointer is associated with a **dummy argument** of the procedure, and
 - (a) the **effective argument** does not have the **TARGET attribute** or is an **array section** with a **vector subscript**, or
 - (b) the **dummy argument** has the **VALUE attribute**,
 - (11) a **BLOCK construct** completes execution and the pointer is an **unsaved construct entity** of that **BLOCK construct**,
 - (12) a **DO CONCURRENT construct** is terminated and the pointer's association status was changed in more than one iteration of the construct,
 - (13) an iteration of a **DO CONCURRENT construct** completes and the pointer is associated with a variable of that construct that has **LOCAL** or **LOCAL_INIT** locality,
 - (14) the pointer is a **subcomponent** of an object that is allocated and either
 - (a) the pointer is not **default-initialized** and **SOURCE=** does not appear, or
 - (b) **SOURCE=** appears and the association status of the corresponding **subcomponent** of the *source-expr* is undefined,

¹⁸Common blocks are obsolescent.

- (15) the pointer is a **subcomponent** of an object, the pointer is not **default-initialized**, and a procedure is invoked with this object as an **actual argument** corresponding to a dummy argument with **INTENT (OUT)**,
- (16) a procedure is invoked with the pointer as an **actual argument** corresponding to a pointer dummy argument with **INTENT (OUT)**, or
- (17) evaluation of an expression containing a function reference that need not be evaluated completes, if execution of that function would change the association status of the pointer.

19.5.2.6 Other events that change the association status of pointers

- 1 When a pointer becomes associated with another pointer by **argument association**, **construct association**, or **host association**, the effects on its association status are specified in 19.5.5.
- 2 While two pointers are **name associated**, **storage associated**, or **inheritance associated**, if the association status of one pointer changes, the association status of the other changes accordingly.
- 3 The association status of a pointer object with the **VOLATILE attribute** might change by means not specified by the program.

19.5.2.7 Pointer definition status

- 1 The definition status of an associated **data pointer** is that of its **target**. If a pointer is associated with a **definable target**, it becomes defined or undefined according to the rules for a variable (19.6). The definition status of a pointer that is not associated is undefined.

19.5.3 Storage association (obsolescent)

19.5.3.1 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**.
- 2 Storage association is established only by **COMMON**, **ENTRY**, and **EQUIVALENCE** statements. All of those statements are obsolescent, and thus the entire concept of storage association is obsolescent.

19.5.3.2 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** (12.3.5), or an **unspecified storage unit**. The sizes of the **numeric storage unit**, the **character storage unit** and the **file storage unit** are the values of constants in the ISO_FORTRAN_ENV intrinsic module (16.10.2).
- 2 In a storage association context
 - (1) a nonpointer scalar object that is default integer, default real, or default logical occupies a single **numeric storage unit**,
 - (2) a nonpointer scalar object that is double precision real or default complex occupies two **contiguous numeric storage units**,

- (3) a default character nonpointer scalar object of character length *len* occupies *len* contiguous character storage units,
- (4) if C character kind is not the same as default character kind a nonpointer scalar object of type character with the C character kind (18.2.2) and character length *len* occupies *len* contiguous unspecified storage units,
- (5) a nonpointer scalar object of sequence type occupies a sequence of storage sequences corresponding to the sequence of its ultimate components,
- (6) a nonpointer scalar object of any type not specified in items (1)-(5) occupies a single unspecified storage unit that is different for each case and each set of type parameter values, and that is different from the unspecified storage units of item (4),
- (7) a nonpointer array occupies a sequence of contiguous storage sequences, one for each array element, in array element order (9.5.3.3), 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 (8.10.3.2).

19.5.3.3 Association of storage sequences

- 1 Two nonzero-sized storage sequences s_1 and s_2 are storage associated if the i th storage unit of s_1 is the same as the j th storage unit of s_2 . This causes the $(i+k)$ th storage unit of s_1 to be the same as the $(j+k)$ th storage unit of s_2 , for each integer k such that $1 \leq i+k \leq \text{size of } s_1$ and $1 \leq j+k \leq \text{size of } s_2$ where *size of* measures the number of storage units.
- 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.

19.5.3.4 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 (8.10.2.2, 8.10.3.2).
- 4 COMMON statements cause data objects in one scoping unit to become storage associated with data objects

in another [scoping unit](#).

A [common block](#) is permitted to contain a sequence of differing [storage units](#). All [scoping units](#) that access named [common blocks](#) with the same name shall specify an identical sequence of [storage units](#). [Blank common blocks](#) may be declared with differing sizes in different [scoping units](#). For any two [blank common blocks](#), the initial sequence of [storage units](#) of the longer [blank common block](#) shall be identical to the sequence of [storage units](#) of the shorter [common block](#). If two [blank common blocks](#) are the same length, they shall have the same sequence of [storage units](#).

An [ENTRY statement](#) in a function subprogram causes storage association of the [function results](#) that are variables.

Partial association shall exist only between

- an object that is default character or of [character sequence type](#) and an object that is default character or of [character sequence type](#), or
- an object that is default complex, double precision real, or of [numeric sequence type](#) and an object that is default integer, default real, default logical, double precision real, default complex, or of [numeric sequence type](#).

For noncharacter entities, partial association shall occur only through the use of [COMMON](#), [EQUIVALENCE](#), or [ENTRY](#) statements. For character entities, partial association shall occur only through [argument association](#) or the use of [COMMON](#) or [EQUIVALENCE](#) statements.

Partial association of character entities occurs when some, but not all, of the [storage units](#) of the entities are the same.

A [storage unit](#) shall not be [explicitly initialized](#) more than once in a program. [Explicit initialization](#) overrides [default initialization](#), and [default initialization](#) for an object of derived type overrides [default initialization](#) for a component of the object (7.5.4.6). [Default initialization](#) may be specified for a [storage unit](#) that is storage associated provided the objects supplying the [default initialization](#) are of the same type and type parameters, and supply the same value for the [storage unit](#).

19.5.4 Inheritance association

[Inheritance association](#) occurs between components of the [parent component](#) and components [inherited](#) by type extension into an [extended type](#) (7.5.7.2). This association is persistent; it is not affected by the accessibility of the [inherited](#) components.

19.5.5 Establishing associations

When an association is established between two entities by [argument association](#), [host association](#), or [construct association](#), certain properties of the [associating entity](#) become those of the pre-existing entity.

For [argument association](#), the pre-existing entity is the [effective argument](#) and the [associating entity](#) is the dummy argument.

For [host association](#), the [associating entity](#) is the entity in the contained [scoping unit](#). When a procedure is invoked, the pre-existing entity that participates in the association is the one from its [host instance](#) (15.6.2.4). Otherwise the pre-existing entity that participates in the association is the entity in the [host scoping unit](#).

- 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.

19.6 Definition and undefinition of variables

19.6.1 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.

19.6.2 Variables that are always defined

- 1 Zero-sized arrays and zero-length strings are always defined.

19.6.3 Variables that are initially defined

The following variables are initially defined:

- (1) variables specified to have initial values by [DATA statements](#);
- (2) variables specified to have initial values by [type declaration statements](#);
- (3) nonpointer [default-initialized subcomponents](#) of [saved](#) variables that do not have the [ALLOC-ATABLE](#) 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

Fortran code:

```
module mod
  integer, bind(c,name="blivet") :: foo
end module mod
```

C code:

```
int blivet = 123;
```

In the above example, the Fortran variable foo is initially defined to have the value 123 by means other than Fortran.

19.6.4 Variables that are initially undefined

Variables that are not initially defined are initially undefined.

19.6.5 Events that cause variables to become defined

Variables become defined by the following events.

- (1) Execution of an [intrinsic assignment statement](#) other than a [masked array assignment](#) or [FORALL](#)¹⁹ assignment statement causes the variable that precedes the equals to become defined.
- (2) Execution of a [masked array assignment](#) or [FORALL](#) assignment statement might cause some or all of the array elements in the [assignment statement](#) to become defined ([10.2.3](#)).
- (3) As execution of an [input statement](#) proceeds, each variable that is assigned a value from the input file becomes defined at the time that data are transferred to it. (See (4) in [19.6.6](#).) Execution of a [WRITE statement](#) whose [unit](#) specifier identifies an [internal file](#) causes each record that is written to become defined.
- (4) Execution of a [DO statement](#) causes the DO variable, if any, to become defined.
- (5) Beginning of execution of the action specified by an [io-implied-do](#) in a synchronous data transfer statement causes the [do-variable](#) to become defined.
- (6) A reference to a procedure causes an entire dummy data object to become defined if the dummy data object does not have [INTENT \(OUT\)](#) and the entire [effective argument](#) is defined.
A reference to a procedure causes a subobject of a dummy argument to become defined if the dummy argument does not have [INTENT \(OUT\)](#) and the corresponding subobject of the [effective argument](#) is defined.
- (7) Execution of an input/output statement containing an [IOSTAT= specifier](#) causes the specified integer variable to become defined.

¹⁹The [FORALL construct](#) and [FORALL statement](#) are obsolescent.

- (8) Execution of a synchronous **input statement** containing a **SIZE= specifier** causes the specified integer variable to become defined.
- (9) Execution of a wait operation (12.7.1) corresponding to an asynchronous **input statement** containing a **SIZE= specifier** causes the specified integer variable to become defined.
- (10) Execution of an **INQUIRE statement** causes any variable that is assigned a value during the execution of the statement to become defined if no error condition exists.
- (11) If an error, end-of-file, or end-of-record condition occurs during execution of an input/output statement that has an **IOMSG= specifier**, the *iomsg-variable* becomes defined.
- (12) When a **character storage unit** becomes defined, all associated **character storage units** become defined.
When a **numeric storage unit** becomes defined, all associated **numeric storage units** of the same type become defined. When an entity of double precision real type becomes defined, all totally associated entities of double precision real type become defined.
When an **unspecified storage unit** becomes defined, all associated **unspecified storage units** become defined.
- (13) When a default complex entity becomes defined, all partially associated default real entities become defined.
- (14) When both parts of a default complex entity become defined as a result of partially associated default real or default complex entities becoming defined, the default complex entity becomes defined.
- (15) When all components of a structure of a **numeric sequence type** or **character sequence type** become defined as a result of partially associated objects becoming defined, the structure becomes defined.
- (16) Execution of a statement with a **STAT= specifier** causes the variable specified by the **STAT= specifier** to become defined.
- (17) If an error condition occurs during execution of a statement that has an **ERRMSG= specifier**, the variable specified by the **ERRMSG= specifier** becomes defined.
- (18) Allocation of a zero-sized array or zero-length character variable causes the array or variable to become defined.
- (19) Allocation of an object that has a nonpointer **default-initialized subcomponent**, except by an **ALLOCATE statement** with a **SOURCE= specifier**, causes that **subcomponent** to become defined.
- (20) Successful execution of an **ALLOCATE statement** with a **SOURCE= specifier** causes a subobject of the allocated object to become defined if the corresponding subobject of the **SOURCE=** expression is defined.
- (21) Invocation of a procedure causes any **automatic data object** of zero size or zero character length in that procedure to become defined.
- (22) When a pointer becomes associated with a **target** that is defined, the pointer becomes defined.
- (23) Invocation of a procedure that contains an **unsaved** nonpointer nonallocatable local variable causes all nonpointer **default-initialized subcomponents** of the object to become defined.
- (24) Invocation of a procedure that has a nonpointer nonallocatable **INTENT (OUT)** dummy argument causes all nonpointer **default-initialized subcomponents** of the dummy argument to become defined.
- (25) In a **DO CONCURRENT** or **FORALL**²⁰ construct, the *index-name* becomes defined when the *index-name* value set is evaluated.

²⁰The **FORALL construct** is obsolescent.

- (26) In a **DO CONCURRENT construct**, a variable with **LOCAL_INIT** locality becomes defined at the beginning of each iteration.
- (27) An object with the **VOLATILE attribute** that is changed by a means not specified by the program might become defined (see 8.5.20).
- (28) Execution of the **BLOCK statement** of a **BLOCK construct** that has an **unsaved** nonpointer nonallocatable local variable causes all nonpointer **default-initialized subcomponents** of the variable to become defined.
- (29) Execution of an **OPEN statement** containing a **NEWUNIT= specifier** causes the specified integer variable to become defined.
- (30) 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.
- (31) Successful execution of a **LOCK statement** that does not contain an **ACQUIRED_LOCK= specifier** causes the **lock variable** to become defined.
- (32) Successful execution of an **UNLOCK statement** causes the **lock variable** to become defined.
- (33) Failure of an **image** that locked a **lock variable** without unlocking it causes the **lock variable** to become defined.
- (34) Successful execution of an **EVENT POST** or **EVENT WAIT** statement causes the **event variable** to become defined.
- (35) Successful execution of a **FORM TEAM statement** causes the **team variable** to become defined.
- (36) Execution of a **FORM TEAM statement** with a **STAT= specifier** that assigns the value **STAT_FAILED_IMAGE** from the intrinsic module **ISO_FORTRAN_ENV** to its **stat-variable** causes the **team variable** to become defined.
- (37) Execution of a **NOTIFY WAIT statement** or an **assignment statement** with a **NOTIFY= specifier** causes the notify variable to become defined.

19.6.6 Events that cause variables to become undefined

1 Variables become undefined by the following events.

- (1) When a scalar variable of **intrinsic type** becomes defined, all totally associated²¹ 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**²² that appears in the subprogram become undefined if they have been defined or redefined, unless another active **scoping unit** is referencing the **common block**, and
 - (c) a variable of type **C_PTR** from the intrinsic module **ISO_C_BINDING** whose value is the **C address** of an **unsaved** local variable of the subprogram becomes undefined.

²¹Storage association is obsolescent.

²²Common blocks are obsolescent.

- (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-dos**, all of the **do-variables** in the statement become undefined (12.11).
- (7) Execution of a direct access **input statement** that specifies a record that has not been written previously causes all of the variables specified by the input list of the statement to become undefined.
- (8) Execution of an **INQUIRE statement** might cause the **NAME=**, **RECL=**, and **NEXTREC=** variables to become undefined (12.10).
- (9) When a **character storage unit** becomes undefined, all associated **character storage units** become undefined.
When a **numeric storage unit** becomes undefined, all associated **numeric storage units** become undefined unless the undefinition is a result of defining an associated **numeric storage unit** of different type (see (1) above).
When an entity of double precision real type becomes undefined, all totally associated entities of double precision real type become undefined.
When an **unspecified storage unit** becomes undefined, all associated **unspecified storage units** become undefined.
- (10) When an **allocatable** entity is deallocated, it becomes undefined.
- (11) When the allocation transfer procedure (16.9.147) causes the allocation status of an **allocatable** entity to become unallocated, the entity becomes undefined.
- (12) Successful execution of an **ALLOCATE statement** with no **SOURCE= specifier** causes a **subcomponent** of an allocated object to become undefined if **default initialization** has not been specified for that **subcomponent**.
- (13) Successful execution of an **ALLOCATE statement** with a **SOURCE= specifier** causes a subobject of the allocated object to become undefined if the corresponding subobject of the **SOURCE=** expression is undefined.
- (14) Execution of an **INQUIRE statement** causes all inquiry specifier variables to become undefined if an error condition exists, except for any variable in an **IOSTAT=** or **IOMSG=** specifier.
- (15) When a procedure is invoked
 - (a) an optional dummy argument that has no corresponding **actual argument** becomes undefined,
 - (b) a dummy argument with **INTENT (OUT)** becomes undefined except for any nonpointer **default-initialized subcomponents** of the argument,
 - (c) an **actual argument** corresponding to a dummy argument with **INTENT (OUT)** becomes undefined except for any nonpointer **default-initialized subcomponents** of the argument,
 - (d) a subobject of a dummy argument that does not have **INTENT (OUT)** becomes undefined if the corresponding subobject of the **effective argument** is undefined, and
 - (e) a variable that is the **function result** of that procedure becomes undefined except for any of its nonpointer **default-initialized subcomponents**.
- (16) When the association status of a pointer becomes undefined or **disassociated** (19.5.2.4, 19.5.2.5), the pointer becomes undefined.

- 1 (17) When a **DO CONCURRENT construct** terminates, a variable that is defined or becomes undefined
2 during more than one iteration of the construct becomes undefined.
- 3 (18) When execution of an iteration of a **DO CONCURRENT construct** completes, a construct entity of
4 that construct which has **LOCAL** or **LOCAL_INIT** locality becomes undefined.
- 5 (19) Execution of an asynchronous **READ statement** causes all of the variables specified by the in-
6 put list or **SIZE= specifier** to become undefined. Execution of an asynchronous namelist **READ**
7 **statement** causes any variable in the namelist group to become undefined if that variable will
8 subsequently be defined during the execution of the **READ statement** or the corresponding wait
9 operation (12.7.1).
- 10 (20) When a variable with the **TARGET attribute** is deallocated, a variable of type **C_PTR** from the
11 intrinsic module **ISO_C_BINDING** becomes undefined if its value is the **C address** of any part of
12 the variable that is deallocated.
- 13 (21) When a pointer is deallocated, a variable of type **C_PTR** from the intrinsic module **ISO_C_BINDING**
14 becomes undefined if its value is the **C address** of any part of the **target** that is deallocated.
- 15 (22) Execution of the allocation transfer procedure (16.9.147) where the argument TO does not have
16 the **TARGET attribute** causes a variable of type **C_PTR** from the intrinsic module **ISO_C_BINDING**
17 to become undefined if its value is the **C address** of any part of the argument FROM.
- 18 (23) When a **BLOCK construct** completes execution,
19
 - its **unsaved** local variables become undefined, and
 - a variable of type **C_PTR** from the intrinsic module **ISO_C_BINDING**, whose value is the **C**
21 **address** of an **unsaved** local variable of the **BLOCK construct**, becomes undefined.
- 22 (24) When execution of the **host instance** of the **target** of a variable of type **C_FUNPTR** from the intrinsic
23 module **ISO_C_BINDING** is completed by execution of a **RETURN** or **END** statement, the variable
24 becomes undefined.
- 25 (25) Execution of an intrinsic assignment of the type **C_PTR** or **C_FUNPTR** from the intrinsic mod-
26 ule **ISO_C_BINDING**, or of the type **TEAM_TYPE** from the intrinsic module **ISO_FORTRAN_ENV**, in
27 which the variable and **expr** are not on the same **image**, causes the variable to become undefined.
- 28 (26) An object with the **VOLATILE attribute** (8.5.20) might become undefined by means not specified
29 by the program.
- 30 (27) When a pointer becomes associated with a target that is undefined, the pointer becomes un-
31 defined.
- 32 (28) When an image fails during execution of a segment, a data object on a nonfailed image becomes
33 undefined if it is not a **lock variable** and it might become undefined by execution of a statement
34 of the segment other than an invocation of an **atomic subroutine** with the object as an **actual**
35 **argument** corresponding to the **ATOM dummy argument**.
- 36 (29) Execution of a **FORM TEAM statement** with a **STAT= specifier** that assigns a nonzero value other
37 than that of **STAT_FAILED_IMAGE** from the intrinsic module **ISO_FORTRAN_ENV** to the *stat-**varia-***
38 ***ble*** causes the **team variable** to become undefined.
- 39 (30) When the **STAT** argument in a reference to a **collective subroutine** is assigned a nonzero value,
40 the **A** argument becomes undefined.
- 41 (31) When an image which references a **collective subroutine** with a present **RESULT_IMAGE** argument
42 is not the **image** identified by **RESULT_IMAGE**, the **A** argument on that **image** becomes undefined.
- 43 (32) When an error condition occurs during execution of an **atomic subroutine** whose **STAT** argument
44 is present, any other argument that is not **INTENT (IN)** becomes undefined.

NOTE

Execution of a [defined assignment statement](#) could leave all or part of the variable undefined.

19.6.7 Variable definition context

1 Some variables are prohibited from appearing in a syntactic context that would imply definition or un-
 2 definition of the variable ([8.5.10](#), [8.5.15](#), [15.7](#)). The following are the contexts in which the appearance of
 3 a variable implies such definition or undefinition of the variable:
 4

- 5 (1) the *variable* of an *assignment-stmt*;
- 6 (2) a *do-variable* in a *do-stmt* or *io-implied-do*;
- 7 (3) an *input-item* in a *read-stmt*;
- 8 (4) a *variable-name* in a *namelist-stmt* if the *namelist-group-name* appears in a *NML= specifier* in a
 9 *read-stmt*;
- 10 (5) an *internal-file-variable* in a *write-stmt*;
- 11 (6) a *SIZE=* or *IOMSG=* specifier in an input/output statement;
- 12 (7) a specifier in an *INQUIRE statement* other than *FILE=*, *ID=*, and *UNIT=*;
- 13 (8) a *NEWUNIT= specifier* in an *OPEN statement*;
- 14 (9) an *allocate-object*, *errmsg-variable*, *notify-variable*, or *stat-variable*;
- 15 (10) an *actual argument* in a reference to a procedure with an *explicit interface* if the corresponding
 16 *dummy argument* is not a pointer and has *INTENT (OUT)* or *INTENT (INOUT)*;
- 17 (11) a *variable* that is a *selector* in an *ASSOCIATE*, *CHANGE TEAM*, *SELECT RANK*, or *SELECT TYPE*
 18 construct if the corresponding *associate name* or any subobject thereof appears in a variable
 19 definition context;
- 20 (12) an *event-variable* in an *EVENT POST* or *EVENT WAIT* statement;
- 21 (13) a *lock-variable* in a *LOCK* or *UNLOCK* statement;
- 22 (14) a *scalar-logical-variable* in an *ACQUIRED_LOCK= specifier*;
- 23 (15) a *team-variable* in a *FORM TEAM* statement.

24 2 If a reference to a function appears in a variable definition context the result of the function reference shall
 25 be a pointer that is associated with a *definable target*. That *target* is the variable that becomes defined or
 26 undefined.

19.6.8 Pointer association context

28 1 Some pointers are prohibited from appearing in a syntactic context that would imply alteration of the
 29 *pointer association* status ([19.5.2.2](#), [8.5.10](#), [8.5.15](#), [15.7](#)). The following are the contexts in which the ap-
 30 pearance of a pointer implies such alteration of its *pointer association* status:

- 31 • a *pointer-object* in a *nullify-stmt*;
- 32 • a *data-pointer-object* or *proc-pointer-object* in a *pointer-assignment-stmt*;
- 33 • an *allocate-object* in an *allocate-stmt* or *deallocate-stmt*;
- 34 • an *actual argument* in a reference to a procedure if the corresponding *dummy argument* is a pointer
 35 with the *INTENT (OUT)* or *INTENT (INOUT)* attribute.

Annex A

(Informative)

Processor dependencies

A.1 Unspecified items

1 This document does not specify the following:

- the properties excluded in 1;
- a processor's error detection capabilities beyond those listed in 4.2;
- which additional intrinsic procedures or modules a processor provides (4.2);
- the number and kind of companion processors (5.5.7);
- the number of representation methods and associated kind type parameter values of the intrinsic types (7.4), except that there shall be at least two representation methods for type real, and a representation method of type complex that corresponds to each representation method for type real.

A.2 Processor dependencies

1 According to this document, the following are processor dependent:

- the order of evaluation of the specification expressions within the specification part of an invoked Fortran procedure (5.3.5);
- how soon an image terminates if another image initiates error termination (5.3.5);
- the value of a reference to a coindexed object on a failed image (5.3.6);
- the conditions that cause an image to fail (5.3.6);
- whether the processor has the ability to detect that an image has failed (5.3.6);
- whether the processor supports a concept of process exit status, and if so, the process exit status on program termination (5.3.7);
- the mechanism of a companion processor, and the means of selecting between multiple companion processors (5.5.7);
- the processor character set (6.1);
- the maximum number of unique statement labels in a program unit (6.2.5);
- the means for specifying the source form of a program unit (6.3);
- in fixed source form, the maximum number of characters allowed on a source line containing characters not of default kind (6.3.3);
- the maximum depth of nesting of include lines (6.4);
- the interpretation of the char-literal-constant in the include line (6.4);
- the set of values supported by an intrinsic type, other than logical (7.1.3);
- the kind type parameter value of a complex literal constant, if both the real part and imaginary part are of type real with the same precision, but have different kind type parameter values (7.4.3.3);
- the kind of a character length type parameter (7.4.4.1);
- the blank padding character for nondefault character kind (7.4.4.2)

- whether particular control characters can appear within a character literal constant in fixed source form (7.4.4.3);
- the [collating sequence](#) for each character set (7.4.4.4);
- the order of [finalization](#) of components of objects of derived type (7.5.6.2);
- the order of [finalization](#) when a single event causes several objects to be finalized (7.5.6.2);
- whether and when an object is finalized if it is allocated by pointer allocation and it later becomes unreachable due to all pointers associated with the object having their [pointer association](#) status changed (7.5.6.3);
- whether an object is finalized by a deallocation in which an error condition occurs (7.5.6.3);
- the kind type parameter of the enumerators of an interoperable enumeration (7.6.1);
- whether an array is contiguous, except as specified in 8.5.7;
- the set of error conditions that can occur in [ALLOCATE](#) and [DEALLOCATE](#) statements (9.7.1, 9.7.3);
- the allocation status of a variable after evaluation of an expression if the evaluation of a function would change the allocation status of the variable and if a reference to the function appears in the expression in which the value of the function is not needed to determine the value of the expression (9.7.1.3);
- the order of deallocation when several objects are deallocated by a [DEALLOCATE statement](#) (9.7.3);
- the order of deallocation when several objects are deallocated due to the occurrence of an event described in 9.7.3.2;
- whether an allocated allocatable subobject is deallocated when an error condition occurs in the deallocation of an object (9.7.3.2);
- the positive integer values assigned to the [stat-variable](#) in a [STAT= specifier](#) as the result of an error condition (9.7.4, 11.7.11);
- the allocation status or [pointer association](#) status of an [allocate-object](#) if an error condition occurs during execution of an [ALLOCATE](#) or [DEALLOCATE](#) statement (9.7.4);
- the value assigned to the [errmsg-variable](#) in an [ERRMSG= specifier](#) as the result of an error condition (9.7.5, 11.7.11);
- the kind type parameter value of the result of a numeric intrinsic binary operation where
 - both operands are of type integer but with different kind type parameters, and the decimal exponent ranges are the same,
 - one operand is of type real or complex and the other is of type real or complex with a different kind type parameter, and the decimal precisions are the same,
 and for a logical intrinsic binary operation where the operands have different kind type parameters (10.1.9.3);
- the character assigned to the variable in an [intrinsic assignment statement](#) if the kind of the expression is different and the character is not representable in the kind of the variable (10.2.1.3);
- the order of evaluation of the [specification expressions](#) within the specification part of a [BLOCK construct](#) when the construct is executed (11.1.4);
- the ordering between records written by different iterations of a [DO CONCURRENT construct](#) if the records are written to a file connected for sequential access by more than one iteration (11.1.7);
- the order in which values are combined in a [DO CONCURRENT reduction](#) (11.1.7.5);
- the manner in which the stop code of a [STOP](#) or [ERROR STOP](#) statement is made available (11.4);
- the value of the count of the [notify variable](#) in a [NOTIFY WAIT statement](#) if an error condition occurs (11.6);
- the mechanisms available for creating dependencies for cooperative synchronization (11.7.5);

- 1 • the value of the count of the event variable in an [EVENT POST](#) or [EVENT WAIT](#) statement if an error
- 2 condition occurs ([11.7.7](#), [11.7.8](#));
- 3 • the [image index](#) value established for each image in a [team](#) by a [FORM TEAM statement](#) without a
- 4 [NEW_INDEX= specifier](#) ([11.7.9](#));
- 5 • the set of error conditions that can occur in [image control statements](#) ([11.7.11](#));
- 6 • the relationship between the [file storage units](#) when viewing a file as a [stream file](#), and the records
- 7 when viewing that file as a [record file](#) ([12](#));
- 8 • whether particular control characters can appear in a formatted [record](#) or a formatted [stream file](#)
- 9 ([12.2.2](#));
- 10 • the form of values in an unformatted record ([12.2.3](#));
- 11 • at any time, the set of allowed access methods, set of allowed forms, set of allowed actions, and set of
- 12 allowed record lengths for a [file](#) ([12.3](#));
- 13 • the set of allowable names for a [file](#) ([12.3](#));
- 14 • whether a named file on one [image](#) is the same as a file with the same name on another [image](#) ([12.3.1](#));
- 15 • the set of [external files](#) that exist for a program ([12.3.2](#));
- 16 • the relationship between positions of successive [file storage units](#) in an [external file](#) that is connected
- 17 for formatted stream access ([12.3.3.4](#));
- 18 • the [external unit](#) preconnected for sequential formatted input and identified by an asterisk or the
- 19 [named constant](#) INPUT_UNIT of the ISO_FORTRAN_ENV intrinsic module ([12.5](#));
- 20 • the [external unit](#) preconnected for sequential formatted output and identified by an asterisk or the
- 21 [named constant](#) OUTPUT_UNIT of the ISO_FORTRAN_ENV intrinsic module ([12.5](#));
- 22 • the [external unit](#) preconnected for sequential formatted output and identified by the [named constant](#)
- 23 ERROR_UNIT of the ISO_FORTRAN_ENV intrinsic module, and whether this unit is the same as OUT-
- 24 PUT_UNIT ([12.5](#));
- 25 • at any time, the set of [external units](#) that exist for an [image](#) ([12.5.3](#));
- 26 • whether a [unit](#) can be connected to a file that is also connected to a C stream ([12.5.4](#));
- 27 • whether a file can be connected to more than one [unit](#) at the same time ([12.5.4](#));
- 28 • the effect of performing input/output operations on multiple units while they are connected to the
- 29 same [external file](#) ([12.5.4](#));
- 30 • the result of performing input/output operations on a [unit](#) connected to a file that is also connected
- 31 to a C stream ([12.5.4](#));
- 32 • whether the files connected to the [units](#) INPUT_UNIT, OUTPUT_UNIT, and ERROR_UNIT correspond
- 33 to the predefined C text streams standard input, standard output, and standard error, respectively
- 34 ([12.5.4](#));
- 35 • the results of performing input/output operations on an [external file](#) both from Fortran and from a
- 36 procedure defined by means other than Fortran ([12.5.4](#));
- 37 • the default value for the [ACTION= specifier](#) in an [OPEN statement](#) ([12.5.6.4](#));
- 38 • the encoding of a file opened with [ENCODING='DEFAULT'](#) ([12.5.6.9](#));
- 39 • the file connected by an [OPEN statement](#) with [STATUS='SCRATCH'](#) ([12.5.6.10](#));
- 40 • the interpretation of case in a file name ([12.5.6.10](#), [12.10.2.2](#));
- 41 • the position of a file after executing an [OPEN statement](#) with a [POSITION= specifier](#) of ASIS, when the
- 42 file previously existed but was not connected ([12.5.6.15](#));
- 43 • the default value for the [RECL= specifier](#) in an [OPEN statement](#) ([12.5.6.16](#));
- 44 • the effect of [RECL=](#) on a record containing any nondefault characters ([12.5.6.16](#));
- 45 • the default input/output rounding mode ([12.5.6.17](#));

- 1 • the default sign mode (12.5.6.18);
- 2 • the file status when `STATUS='UNKNOWN'` is specified in an `OPEN` statement (12.5.6.19);
- 3 • the value assigned to the variable in the `ID= specifier` in an asynchronous `data transfer statement`
- 4 when execution of the statement is successfully completed (12.6.2.9);
- 5 • whether `POS=` is permitted with particular files, and whether `POS=` can position a particular file to a
- 6 position prior to its current position (12.6.2.12);
- 7 • the form in which a single value of derived type is treated in an unformatted input/output statement
- 8 if the `effective item` is not processed by a `defined input/output procedure` (12.6.3);
- 9 • the result of unformatted input when the type or `type parameters` of the value stored in the file differ
- 10 from those of the corresponding `effective item` (12.6.4.5.2);
- 11 • the negative value of the `unit` argument to a `defined input/output procedure` if the parent `data trans-`
- 12 `fer statement` accesses an `internal file` (12.6.4.8.2);
- 13 • the manner in which the processor makes the value of the `iomsg` argument of a `defined input/out-`
- 14 `put procedure` available if the procedure assigns a nonzero value to the `iostat` argument and the
- 15 processor therefore `terminates execution` of the program (12.6.4.8.2);
- 16 • the action caused by the flush operation, whether the processor supports the flush operation for
- 17 the specified `unit`, and the negative value assigned to the `IOSTAT=` variable if the processor does not
- 18 support the flush operation for the specified `unit` (12.9);
- 19 • the case of characters assigned to the variable in a `NAME= specifier` in an `INQUIRE statement`
- 20 (12.10.2.16);
- 21 • which of the connected external unit numbers is assigned to the `scalar-int-variable` in the `NUMBER=`
- 22 `specifier` in an `INQUIRE` by file statement, if more than one unit on an `image` is connected to the file
- 23 (12.10.2.19);
- 24 • the value of the variable in a `POSITION= specifier` in an `INQUIRE statement` if the file has been reposi-
- 25 tioned since connection (12.10.2.24);
- 26 • the relationship between file size and the data stored in records in a sequential or direct access file
- 27 (12.10.2.31);
- 28 • the number of `file storage units` needed to store data in an unformatted file (12.10.3);
- 29 • the set of error conditions that can occur in input/output statements (12.11.1);
- 30 • when an input/output error condition occurs or is detected (12.11.1);
- 31 • the positive integer value assigned to the variable in an `IOSTAT= specifier` as the result of an error
- 32 condition (12.11.5);
- 33 • the value assigned to the variable in an `IOMSG= specifier` as the result of an error condition (12.11.6);
- 34 • the result of output of non-representable characters to a Unicode file (13.7.1);
- 35 • the interpretation of the optional non-blank characters within the parentheses of a real NaN input
- 36 field (13.7.2.3.2);
- 37 • the interpretation of a sign in a NaN input field (13.7.2.3.2);
- 38 • for output of an IEEE NaN, whether after the letters 'NaN', the processor produces additional alpha-
- 39 numeric characters enclosed in parentheses (13.7.2.3.2);
- 40 • the choice of binary exponent in EX output editing (13.7.2.3.6);
- 41 • the effect of the input/output rounding mode `PROCESSOR_DEFINED` (13.7.2.3.8);
- 42 • which value is chosen if the input/output rounding mode is `NEAREST` and the value to be converted
- 43 is exactly halfway between the two nearest representable values in the result format (13.7.2.3.8);
- 44 • the field width, decimal part width, and exponent width used for the G0 edit descriptor (13.7.5);
- 45 • the file position when position editing skips a character of nondefault kind in an `internal file` of default

- 1 character kind or an [external unit](#) that is not connected to a Unicode file ([13.8.1.1](#));
- 2 • when the sign mode is PROCESSOR_DEFINED, whether a plus sign appears in a numeric output field
- 3 for a nonnegative value ([13.8.4](#));
- 4 • whether a leading zero is produced when the leading zero mode is PROCESSOR_DEFINED ([13.8.5](#));
- 5 • the results of list-directed output ([13.10.4](#));
- 6 • the results of namelist output ([13.11.4](#));
- 7 • the interaction between [argument association](#) and [pointer association](#) ([15.5.2.5](#));
- 8 • the values returned by some intrinsic functions ([16](#));
- 9 • how the sequences of atomic actions in unordered segments interleave ([16.5](#));
- 10 • the value assigned to a STAT argument in a reference to an atomic subroutine when an error condition
- 11 occurs ([16.5](#));
- 12 • the effect of calling EXECUTE_COMMAND_LINE on any [image](#) other than [image 1](#) in the [initial team](#)
- 13 ([16.7](#));
- 14 • whether the results returned from CPU_TIME, DATE_AND_TIME and SYSTEM_CLOCK are dependent
- 15 on which [image](#) calls them ([16.7](#));
- 16 • the set of error conditions that can occur in some intrinsic subroutines ([16.9](#));
- 17 • the value assigned to a CMDSTAT, ERRMSG, EXITSTAT, STAT, or STATUS argument to indicate a proces-
- 18 sor-dependent error condition ([16.9](#));
- 19 • the computed value of the intrinsic subroutine CO_REDUCE ([16.9.57](#)) and the intrinsic subroutine
- 20 CO_SUM ([16.9.58](#));
- 21 • whether command arguments are available ([16.9.59](#), [16.9.93](#));
- 22 • the value assigned to the TIME argument by the intrinsic subroutine CPU_TIME ([16.9.67](#));
- 23 • whether date, clock, and time zone information is available ([16.9.69](#));
- 24 • whether date, clock, and time zone information on one [image](#) is the same as that on another [image](#)
- 25 ([16.9.69](#));
- 26 • whether asynchronous command line execution is available ([16.9.83](#));
- 27 • whether the program invocation command is available ([16.9.92](#));
- 28 • the value of command argument zero, if the processor does not support the concept of a command
- 29 name ([16.9.93](#));
- 30 • the order of command arguments ([16.9.93](#));
- 31 • whether the significant length of a command argument includes trailing blanks ([16.9.93](#));
- 32 • the interpretation of case for the NAME argument of the intrinsic subroutine GET_ENVIRONMENT_-
- 33 VARIABLE ([16.9.94](#));
- 34 • whether an environment variable that exists on an [image](#) also exists on another [image](#), and if it does
- 35 exist on both [images](#), whether the values are the same or different ([16.9.94](#));
- 36 • the value assigned to the pseudorandom number seed by the intrinsic subroutine RANDOM_INIT
- 37 ([16.9.167](#));
- 38 • the computation of the seed value used by the pseudorandom number generator ([16.9.169](#));
- 39 • the value assigned to the seed by the intrinsic subroutine RANDOM_SEED when no argument is
- 40 present ([16.9.169](#));
- 41 • the values assigned to its arguments by the intrinsic subroutine SYSTEM_CLOCK ([16.9.202](#));
- 42 • the values of the named constants in the intrinsic module ISO_FORTRAN_ENV ([16.10.2](#));
- 43 • the values returned by the functions COMPILER_OPTIONS and COMPILER_VERSION in the intrinsic
- 44 module ISO_FORTRAN_ENV ([16.10.2](#));

- the extent to which a processor supports IEEE arithmetic (17);
- whether a flag that is quiet on entry to a scoping unit that does not access IEEE_FEATURES, IEEE_EXCEPTIONS, or IEEE_ARITHMETIC is signaling on exit (17.1);
- the conditions under which IEEE_OVERFLOW is raised in a calculation involving non-ISO/IEC/IEEE 60559:2020 floating-point data (17.3);
- the conditions under which IEEE_OVERFLOW and IEEE_DIVIDE_BY_ZERO are raised in a floating-point exponentiation operation (17.3);
- the conditions under which IEEE_DIVIDE_BY_ZERO is raised in a calculation involving floating-point data that do not conform to ISO/IEC/IEEE 60559:2020 (17.3);
- whether an exception signals at the end of a sequence of statements that has no invocations of IEEE_GET_FLAG, IEEE_SET_FLAG, IEEE_GET_STATUS, IEEE_SET_STATUS, or IEEE_SET_HALTING_MODE, in which execution of an operation would cause it to signal, if no value of a variable depends upon the result of the operation (17.3);
- the initial rounding modes (17.4);
- whether the processor supports a particular rounding mode (17.4);
- the effect of the rounding mode IEEE_OTHER, if supported (17.4);
- the initial underflow mode (17.5);
- the initial halting mode (17.6);
- whether IEEE_INT implements the convertToInteger{round} or convertToIntegerExact{round} operation specified by ISO/IEC 60559:2020 (17.11.11);
- which argument is the result value of IEEE_MAX_NUM, IEEE_MAX_NUM_MAG, IEEE_MIN_NUM, or IEEE_MIN_NUM_MAG when both of the arguments are quiet NaNs or are zeros (17.11.19, 17.11.20, 17.11.23, 17.11.24);
- the requirements on the storage sequence to become associated with the pointer FPTR by the C_F_POINTER subroutine (18.2.3.4);
- the order of the members of the CFI_dim_t structure defined in the source file CFI_Fortran_binding.h (18.5.2);
- members of the CFI_cdesc_t structure defined in the source file CFI_Fortran_binding.h beyond the requirements of 18.5.3;
- the value of CFI_MAX_RANK in the source file CFI_Fortran_binding.h (18.5.4);
- the value of CFI_VERSION in the source file CFI_Fortran_binding.h (18.5.4);
- which error condition is detected if more than one error condition could be detected for an invocation of one of the functions declared in the source file CFI_Fortran_binding.h (18.5.5.1);
- the values of the attribute specifier macros defined in the source file CFI_Fortran_binding.h (18.5.4);
- the values of the type specifier macros defined in the source file CFI_Fortran_binding.h;
- which additional type specifier values are defined in the source file CFI_Fortran_binding.h (18.5.4);
- the values of the error code macros other than CFI_SUCCESS that are defined in the source file CFI_Fortran_binding.h (18.5.4);
- the base address of a zero-sized array (18.5.3);
- the values of the floating-point exception flags on entry to a procedure defined by means other than Fortran (18.10.3);
- whether a procedure defined by means other than Fortran is an asynchronous communication initiation or completion procedure (18.10.4).

Annex B

(Informative)

Deleted and obsolescent features

B.1 Deleted features from Fortran 90

1 These deleted features are those features of Fortran 90 that were redundant and considered largely unused.

2 The following Fortran 90 features are not required.

- (1) Real and double precision DO variables.
In FORTRAN 77 and Fortran 90, a DO variable was allowed to be of type real or double precision in addition to type integer; this has been deleted. A similar result can be achieved by using a **DO construct** with no loop control and the appropriate exit test.
- (2) Branching to an **END IF statement** from outside its block.
In FORTRAN 77 and Fortran 90, it was possible to branch to an **END IF statement** from outside the **IF construct**; this has been deleted. A similar result can be achieved by branching to a **CONTINUE statement** that is immediately after the **END IF statement**.
- (3) PAUSE statement.
The PAUSE statement, provided in FORTRAN 66, FORTRAN 77, and Fortran 90, has been deleted. A similar result can be achieved by writing a message to the appropriate **unit**, followed by reading from the appropriate **unit**.
- (4) ASSIGN and assigned GO TO statements, and assigned format specifiers.
The ASSIGN statement and the related assigned GO TO statement, provided in FORTRAN 66, FORTRAN 77, and Fortran 90, have been deleted. Further, the ability to use an assigned integer as a format, provided in FORTRAN 77 and Fortran 90, has been deleted. A similar result can be achieved by using other control constructs instead of the assigned GO TO statement and by using a default character variable to hold a format specification instead of using an assigned integer.
- (5) H edit descriptor.
In FORTRAN 77 and Fortran 90, there was an alternative form of character string edit descriptor, which had been the only such form in FORTRAN 66; this has been deleted. A similar result can be achieved by using a character string edit descriptor.
- (6) Vertical format control.
In FORTRAN 66, FORTRAN 77, Fortran 90, and Fortran 95 formatted output to certain **units** resulted in the first character of each record being interpreted as controlling vertical spacing. There was no standard way to detect whether output to a **unit** resulted in this vertical format control, and no way to specify that it needs to 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 60559:2020 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 that were redundant in a previous standard, and for which better methods were available in that standard. The nature of the obsolescent features is described in [4.4.3](#). The obsolescent features in this document are the following.
 - (1) Alternate return — see [B.3.2](#).
 - (2) [Computed GO TO](#) — see [B.3.3](#).
 - (3) Statement functions — see [B.3.4](#).
 - (4) [DATA statements](#) amongst executable statements — see [B.3.5](#).
 - (5) Assumed length character functions — see [B.3.6](#).
 - (6) Fixed form source — see [B.3.7](#).
 - (7) CHARACTER* form of CHARACTER declaration — see [B.3.8](#).
 - (8) [ENTRY statements](#) — see [B.3.9](#).
 - (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

```

1      CALL SUBR_NAME (X, Y, Z, RETURN_CODE)
2      SELECT CASE (RETURN_CODE)
3          CASE (1)
4              ...
5          CASE (2)
6              ...
7          CASE (3)
8              ...
9          CASE DEFAULT
10             ...
11      END SELECT

```

12 B.3.3 Computed GO TO statement

- 13 1 The [computed GO TO statement](#) has been superseded by the [SELECT CASE construct](#), which is a generalized,
 14 easier to use, and clearer means of expressing the same computation.

15 B.3.4 Statement functions

- 16 1 [Statement functions](#) are subject to a number of nonintuitive restrictions and are a potential source of error
 17 because their syntax is easily confused with that of an assignment statement.
- 18 2 The internal function is a more generalized form of the [statement function](#) and completely supersedes it.

19 B.3.5 DATA statements among executables

- 20 1 The statement ordering rules allow [DATA statements](#) to appear anywhere in a [program unit](#) after the spe-
 21 cification statements. The ability to position [DATA statements](#) amongst executable statements is very rarely
 22 used, unnecessary, and a potential source of error.

23 B.3.6 Assumed character length functions

- 24 1 Assumed character length for functions is an irregularity in the language in that elsewhere in Fortran the
 25 philosophy is that the attributes of a function result depend only on the [actual arguments](#) of the invocation
 26 and on any data accessible by the function through host or use association. Some uses of this facility can be
 27 replaced with an [automatic](#) character length function, where the length of the function result is declared
 28 in a [specification expression](#). Other uses can be replaced by the use of a subroutine whose arguments
 29 correspond to the function result and the function arguments.
- 30 2 Note that dummy arguments of a function can have assumed character length.

31 B.3.7 Fixed form source

- 32 1 Fixed form source was designed when the principal machine-readable input medium for new programs
 33 was punched cards. Now that new and amended programs are generally entered via keyboards with screen
 34 displays, it is an unnecessary overhead, and is potentially error-prone, to have to locate positions 6, 7, or
 35 72 on a line. Free form source was designed expressly for this more modern technology.
- 36 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 Feature history

C.1.1 Features that were new in Fortran 2023

- Source form:
The maximum length of a line in free form source has been increased. The maximum length of a statement has been increased. The limit on the number of continuation lines has been removed.
- Data declaration:
A data object with a coarray component can be an array or allocatable. BIND(C) ENUM are now referred to as interoperable enumerations, and noninteroperable enumeration types are available. An interoperable enumeration can be given a type name. TYPEOF and CLASSOF type specifiers can be used to declare one or more entities to have the same type and type parameters as another entity. A PUBLIC namelist group can have a PRIVATE namelist group object. The DIMENSION attribute can be declared with a syntax that does not depend on the rank (8.5.8, 8.5.17).
- Data usage and computation:
Binary, octal, and hexadecimal literal constants can be used in additional contexts. A deferred-length allocatable *errmsg-variable* is allocated by the processor to the length of the explanatory message. An ALLOCATE statement can specify the bounds of an array allocation with array expressions. A pointer assignment statement can specify lower bounds or rank remapping with array expressions. Arrays can be used to specify multiple subscripts or subscript triplets (9.5.3.2). Conditional expressions provide selective evaluation of subexpressions.
- Input/output:
The AT edit descriptor provides output of character values with trailing blanks trimmed. The LEADING_ZERO= specifier in the OPEN and WRITE statements, and the LZF, LZS and LZ control edit descriptors, provide control of optional leading zeros during formatted output. A deferred-length allocatable *iomsg-variable* is allocated by the processor to the length of the explanatory message. A deferred-length allocatable scalar *io-unit* in a WRITE statement is allocated by the processor to the length of the record to be written.
- Execution control:
The REDUCE locality specifier for the DO CONCURRENT construct specifies reduction variables for the loop. The NOTIFY WAIT statement, NOTIFY= specifier on an image selector, and the NOTIFY-TYPE from the intrinsic module ISO_FORTRAN_ENV provide one-sided data-oriented synchronization between images.
- Intrinsic procedures:
The intrinsic functions ACOSD, ASIND, ATAND, ATAN2D, COSD, SIND, and TAND are trigonometric functions in which angles are specified in degrees. The intrinsic functions ACOSPI, ASINPI, ATANPI, ATAN2PI, COSPI, SINPI, and TANPI are trigonometric functions in which angles are specified in half-revolutions (that is, as multiples of π). The intrinsic function SELECTED_LOGICAL_KIND returns kind type parameter values for type logical. The intrinsic subroutine SPLIT parses a string into tokens, one at a time. The intrinsic subroutine SYSTEM_CLOCK supports more than one system clock for an image.

The intrinsic subroutine `TOKENIZE` parses a string into tokens. When a `deferred-length` allocatable actual argument of an intrinsic procedure is assigned character data, it is allocated by the processor to the length of the data. Execution of a `collective subroutine` can be successful on an `image` even when an error condition occurs for the corresponding execution on another `image`.

- Intrinsic modules:

Additional named constants `LOGICAL8`, `LOGICAL16`, `LOGICAL32`, `LOGICAL64`, and `REAL16` have been added to the intrinsic module `ISO_FORTRAN_ENV`. The subroutines `IEEE_GET_ROUNDING_MODE`, `IEEE_GET_UNDERFLOW_MODE`, `IEEE_SET_ROUNDING_MODE`, and `IEEE_SET_UNDERFLOW_MODE`, from the intrinsic module `IEEE_ARITHMETIC`, are now considered to be pure and simple. The subroutines `IEEE_GET_MODES`, `IEEE_GET_STATUS`, `IEEE_SET_MODES`, and `IEEE_SET_STATUS`, from the intrinsic module `IEEE_EXCEPTIONS`, are now considered to be pure and simple. The procedures `C_F_STRPTR` and `F_C_STRING` have been added to the intrinsic module `ISO_C_BINDING` to assist in the use of null-terminated strings. The subroutine `C_F_POINTER` in the intrinsic module `ISO_C_BINDING` has an extra optional dummy argument, `LOWER`, that specifies the lower bounds for `FPTR`.

- Changes to the intrinsic module `IEEE_ARITHMETIC` for conformance with ISO/IEC 60559:2020:

The new functions `IEEE_MAX`, `IEEE_MAX_MAG`, `IEEE_MIN`, and `IEEE_MIN_MAG` perform the operations maximum, maximumMagnitude, minimum, and minimumMagnitude in ISO/IEC 60559:2020. The functions `IEEE_MAX_NUM`, `IEEE_MAX_NUM_MAG`, `IEEE_MIN_NUM`, and `IEEE_MIN_NUM_MAG` now conform to the operations maximumNumber, maximumMagnitudeNumber, minimumNumber and minimumMagnitudeNumber in ISO/IEC 60559:2020; the changes affect the treatment of zeros and NaNs. The result of the function `IEEE_SCALB` from the intrinsic module `IEEE_ARITHMETIC` has been corrected to conform to ISO/IEC 60559:2020; the function has also been changed to implement the `scaleB` operation.

- Program units and procedures:

A procedure can be specified to be a `simple procedure`; a `simple procedure` references or defines nonlocal variables only via its `dummy arguments`. `Conditional arguments` provide `actual argument` selection in a `procedure reference`.

C.1.2 Features that were new in Fortran 2018

- Data declaration:

Constant properties of an object declared in its `entity-decl` can be used in its `initialization`. The `EQUIVALENCE` and `COMMON` statements and the `block data program unit` have been redundant since Fortran 90 and are now specified to be `obsolescent`. Diagnosis of the appearance of a `PROTECTED TARGET` variable accessed by `use association` as a `data-target` in a structure constructor is required.

- Data usage and computation:

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

- Input/output:

The `SIZE= specifier` can be used with advancing input. It is no longer prohibited to open a file on

more than one unit. The value assigned by the **RECL= specifier** in an **INQUIRE statement** has been standardized. The values assigned by the **POS=** and **SIZE=** specifiers in an **INQUIRE statement** for a unit that has pending asynchronous operations have been standardized. The **G0.d edit descriptor** can be used for **effective 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 60559:2020. The **EX edit descriptor** provides hexadecimal-significand formatted output conforming to ISO/IEC 60559:2020. An error condition occurs if unacceptable characters are presented for logical or numeric editing during execution of a formatted input statement.

- Execution control:

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

- Intrinsic procedures and modules:

In a reference to the intrinsic function **CMPLX** with an actual argument of type complex, no keyword is needed for a **KIND** argument. 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**. 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**. All standard procedures in the intrinsic module **ISO_C_BINDING**, other than **C_F_POINTER** and **C_F_PROCPONTER**, are now **pure**. The arguments to the intrinsic function **SIGN** can be of different kind. Nonpolymorphic pointer arguments to the intrinsic functions **EXTENDS_TYPE_OF** and **SAME_TYPE_AS** need not have defined **pointer association** status. The effects of invoking the intrinsic procedures **COMMAND_ARGUMENT_COUNT**, **GET_COMMAND**, and **GET_COMMAND_ARGUMENT**, on **images** other than **image** one, are no longer processor dependent. Access to error messages from the intrinsic subroutines **GET_COMMAND**, **GET_COMMAND_ARGUMENT**, and **GET_ENVIRONMENT_VARIABLE** is provided by an optional **ERRMSG** argument. The result of **NORM2** for a zero-sized array argument has been clarified.

- Program units and procedures:

The **IMPORT statement** can appear in a contained subprogram or **BLOCK construct**, and can restrict access via host association; diagnosis of violation of the **IMPORT** restrictions is required. The **GENERIC statement** can be used to declare generic interfaces. The number of procedure arguments is used in **generic resolution**. In a module, the **default accessibility** of entities accessed from another module can be controlled separately from the **default accessibility** of entities declared in the using module. An **IMPLICIT NONE statement** can require explicit declaration of the **EXTERNAL attribute** throughout a **scoping unit** and its contained **scoping units**. A **defined operation** need not specify **INTENT (IN)** for a **dummy argument** with the **VALUE**. A **defined assignment** need not specify **INTENT (IN)** for the second **dummy argument** if it has the **VALUE**. Procedures that are not declared with an

asterisk *type-param-value*, 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. The *ERROR STOP statement* can appear in a *pure subprogram*. A *dummy argument* of a *pure function* is permitted in a variable definition context, if it has the *VALUE attribute*. A *coarray dummy argument*, or a *coarray ultimate component* of a *dummy argument*, can be referenced or defined by another *image*.

- 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. The argument to *C_LOC* can be a noninteroperable array. The FPTR argument to *C_F_POINTER* can be a noninteroperable *array pointer*. The argument to *C_FUNLOC* can be a noninteroperable procedure. The FPTR argument to *C_F_PROCPONTER* can be a noninteroperable *procedure pointer*. There is a new *named constant* *C_PTRDIFF_T* to provide interoperability with the C type *ptrdiff_t*.

Additionally to ISO/IEC TS 29113:2012, a scalar *actual argument* can be associated with an *assumed-type assumed-size dummy argument*, an *assumed-rank dummy data object* that is not associated with an *assumed-size array* can be used as the argument to the function *C_SIZEOF* from the intrinsic module *ISO_C_BINDING*, and the *type* argument to *CFI_establish* can have a positive value corresponding to an interoperable C type.

- Changes to the intrinsic modules *IEEE_ARITHMETIC*, *IEEE_EXCEPTIONS*, and *IEEE_FEATURES* for conformance with ISO/IEC 60559:2020:

There is a new, optional, rounding mode *IEEE_AWAY*. The new type *IEEE_MODES_TYPE* encapsulates all floating-point modes. Features associated with subnormal numbers can be accessed with functions and types named *...SUBNORMAL...* (the old *...DENORMAL...* names remain). The new function *IEEE_FMA* performs fused multiply-add operations. The function *IEEE_INT* performs rounded conversions to integer type. The new functions *IEEE_MAX_NUM*, *IEEE_MAX_NUM_MAG*, *IEEE_MIN_NUM*, and *IEEE_MIN_NUM_MAG* calculate maximum and minimum numeric values. The new functions *IEEE_NEXT_DOWN* and *IEEE_NEXT_UP* return the adjacent machine numbers. The new functions *IEEE_QUIET_EQ*, *IEEE_QUIET_GE*, *IEEE_QUIET_GT*, *IEEE_QUIET_LE*, *IEEE_QUIET_LT*, and *IEEE_QUIET_NE* perform quiet comparisons. The new functions *IEEE_SIGNALING_EQ*, *IEEE_SIGNALING_GE*, *IEEE_SIGNALING_GT*, *IEEE_SIGNALING_LE*, *IEEE_SIGNALING_LT*, and *IEEE_SIGNALING_NE* perform signaling comparisons. The decimal rounding mode can be inquired and set independently of the binary rounding mode, using the *RADIX* argument to *IEEE_GET_ROUNDING_MODE* and *IEEE_SET_ROUNDING_MODE*. The new function *IEEE_REAL* performs rounded conversions to real type. The function *IEEE_REM* now requires its arguments to have the same radix. 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.

- Features previously described by ISO/IEC TS 18508:2015:

The *CRITICAL statement* has optional *ERRMSG=* and *STAT=* specifiers. The intrinsic subroutines *ATOMIC_DEFINE* and *ATOMIC_REF* have an optional *STAT* argument. The new intrinsic subroutines *ATOMIC_ADD*, *ATOMIC_AND*, *ATOMIC_CAS*, *ATOMIC_FETCH_ADD*, *ATOMIC_FETCH_AND*, *ATOMIC_FETCH_OR*, *ATOMIC_FETCH_XOR*, *ATOMIC_OR*, and *ATOMIC_XOR* perform atomic operations. The new intrinsic functions *FAILED_IMAGES* and *STOPPED_IMAGES* return indices of images known to have *failed* or *stopped* respectively. The new intrinsic function *IMAGE_STATUS* returns the image execution status of an image. The intrinsic subroutine *MOVE_ALLOC* has optional *ERRMSG* and *STAT*

arguments. The intrinsic functions `IMAGE_INDEX` and `NUM_IMAGES` have additional forms with a `TEAM` or `TEAM_NUMBER` argument. The intrinsic function `THIS_IMAGE` has an optional `TEAM` argument. The `EVENT POST` and `EVENT WAIT` statements, the intrinsic subroutine `EVENT_QUERY`, and the type `EVENT_TYPE` provide an event facility for one-sided segment ordering. The `CHANGE TEAM construct`, derived type `TEAM_TYPE`, `FORM TEAM` and `SYNC TEAM` statements, intrinsic functions `GET_TEAM` and `TEAM_NUMBER`, and the `TEAM=` and `TEAM_NUMBER=` specifiers on image selectors, provide a team facility for a subset of the program's images to act in concert as if it were the set of all images. This team facility allows an allocatable `coarray` to be allocated or deallocated on a subset of images. The new intrinsic subroutines `CO_BROADCAST`, `CO_MAX`, `CO_MIN`, `CO_REDUCE`, and `CO_SUM` perform collective reduction operations on the images of the `current team`. The concept of `failed images`, the `FAIL IMAGE statement`, the `STAT=` specifier on image selectors, and the `named constant STAT_FAILED_IMAGE` from the intrinsic module `ISO_FORTRAN_ENV` provide support for fault-tolerant parallel execution.

- Changes to features previously described by ISO/IEC TS 18508:2015:
The `CHANGE TEAM` and `SYNC TEAM` statements, and the `TEAM=` specifier on image selectors, permit the team to be specified by an expression. The intrinsic functions `FAILED_IMAGES` and `STOPPED_IMAGES` have no restriction on the kind of their result. The name of the function argument to the intrinsic function `CO_REDUCE` is `OPERATION` instead of `OPERATOR`; this argument is not required to be commutative. The `named constant STAT_UNLOCKED_FAILED_IMAGE` from the intrinsic module `ISO_FORTRAN_ENV` indicates that a `lock variable` was locked by an image that `failed`. The `team number` for the `initial team` can be used in image selectors, and in the intrinsic functions `NUM_IMAGES` and `IMAGE_INDEX`. A `team variable` that appears in a `CHANGE TEAM statement` can no longer be defined or become undefined during execution of the `CHANGE TEAM construct`. All `images` of the `current team` are no longer required to execute the same `CHANGE TEAM statement`. A variable of type `TEAM_TYPE` from the intrinsic module `ISO_FORTRAN_ENV` is not permitted to be a `coarray`. A variable of type `TEAM_TYPE` from the intrinsic module `ISO_FORTRAN_ENV` can have a pointer component, and a `team variable` becomes undefined if assigned a value from another `image`. The intrinsic function `UCOBOUND` produces a value for the final upper `cobound` that is always relative to the `current team`. An `EXIT statement` can be used to complete execution of a `CHANGE TEAM` or `CRITICAL construct`.

C.2 Features that were new in Fortran 2008

- Module enhancements:
Submodules provide additional structuring facilities for modules. Data objects and `procedure pointers` declared in a module implicitly have the `SAVE attribute`.
- Parallel execution:
`Coarrays` and synchronization constructs support parallel programming using a single program multiple data (SPMD) model.
- Performance enhancements:
The `DO CONCURRENT construct` provides a means for the program to specify that individual loop iterations have no interdependencies. The `CONTIGUOUS attribute` provides a means for the program to specify restrictions on the storage layout of pointer targets and assumed-shape dummy arguments.
- Data declaration:
The maximum rank has been increased to 15. A processor is required to support at least one kind of integer with a range of at least 18 decimal digits. An `allocatable` component can be of recursive type. A `named constant` array's shape can be implied by its value. A pointer can be initially associated with a target. Subscripts and nested implied-do limits inside a `data-implied-do` can be any

constant expression instead of being limited to combinations of constants, implied-do variables, and intrinsic operations. A FORALL index variable can have its type and kind explicitly declared within the construct. The TYPE keyword can be used to declare entities of intrinsic type. Multiple [type-bound procedures](#) can be declared in a single [type-bound procedure statement](#). An array or object with a nonconstant [length type parameter](#) can have the [VALUE attribute](#). A [PROCEDURE statement](#) can have a double colon before the first procedure name. The [PROTECTED attribute](#) can be specified by the [procedure declaration statement](#).

- Data usage and computation:

A [structure constructor](#) can omit the value for an [allocatable](#) component. [SOURCE=](#) in an [ALLOCATE statement](#) can give an array variable the bounds as well as the value of an expression. [MOLD=](#) in an [ALLOCATE statement](#) can give a [polymorphic](#) variable the shape, type, and type parameters of an expression without copying the value. The real and imaginary parts of a complex entity can be accessed independently with a component-like syntax. [intrinsic assignment](#) Intrinsic assignment to an allocatable polymorphic variable is allowed. A pointer function reference can denote a variable in any variable definition context. Some restrictions on the use of dummy arguments in [elemental subprograms](#) have been removed. Multiple allocations are permitted in a single [ALLOCATE statement](#) with the [SOURCE= specifier](#). A [defined-operator](#) can be used in a [specification expression](#). Any transformational function from the intrinsic module [ISO_C_BINDING](#) can be used in a [specification expression](#).

- Input/output:

[NEWUNIT=](#) in an [OPEN statement](#) automatically selects a unit number that does not interfere with other unit numbers selected by the program. The G0 edit descriptor and unlimited format control ease writing output in comma-separated-value (CSV) format. Input/output of the internal representation of real and complex entities as bit strings can be done using the B, O, and Z edit descriptors ([13.7.2.4](#)). Recursive data transfers are allowed on distinct [units](#).

- Execution control:

The [BLOCK construct](#) can contain declarations of objects with construct scope. The [EXIT statement](#) can transfer control from within more named executable constructs. The [STOP statement](#) has been changed to accept a constant expression instead of merely a literal constant, and to encourage the processor to provide the integer stop code (if it appears) as a termination status (where that makes sense). The [ERROR STOP statement](#) initiates error termination.

- Intrinsic procedures:

- The intrinsic functions [ACOS](#), [ASIN](#), [ATAN](#), [COSH](#), [SINH](#), [TAN](#), and [TANH](#) can have arguments of type complex.
- The new intrinsic functions [ACOSH](#), [ASINH](#), and [ATANH](#) calculate the inverse hyperbolic cosine, sine, and tangent respectively.
- The intrinsic function [ATAN2](#) can be referenced by the name ATAN.
- The new intrinsic subroutines [ATOMIC_DEFINE](#) and [ATOMIC_REF](#) define and reference a variable atomically.
- The new intrinsic functions [BESSEL_J0](#), [BESSEL_J1](#), [BESSEL_JN](#), [BESSEL_Y0](#), [BESSEL_Y1](#), and [BESSEL_YN](#) calculate Bessel functions.
- The new intrinsic functions [BGE](#), [BGT](#), [BLE](#), and [BLT](#) perform bitwise comparisons.
- The new intrinsic functions [DSHIFTL](#) and [DSHIFTR](#) calculate combined left and right shifts.
- The new intrinsic functions [ERF](#), [ERFC](#), and [ERFC_SCALED](#) calculate the error function and its complement.
- The new intrinsic subroutine [EXECUTE_COMMAND_LINE](#) allows a program to start another program.

- The new intrinsic function `FINDLOC` searches an array for a value.
 - The intrinsic functions `LGE`, `LGT`, `LLE`, and `LLT` can have arguments of ASCII kind.
 - The new intrinsic functions `GAMMA` and `LOG_GAMMA` calculate the gamma function and its log.
 - The new intrinsic function `HYPOT` calculates the Euclidean distance.
 - The new intrinsic functions `IALL`, `IANY`, and `IPARITY` reduce an array with the bitwise AND, bitwise OR, and bitwise exclusive OR functions respectively.
 - The new intrinsic function `IMAGE_INDEX` converts cosubscripts to an image index.
 - The new intrinsic functions `LCBOUND` and `UCBOUND` return the cobounds of a coarray.
 - The new intrinsic functions `LEADZ` and `TRAILZ` return the number of leading and trailing zero bits in an integer.
 - The new intrinsic functions `MASKL` and `MASKR` return simple left and right justified masks.
 - A `BACK=` argument has been added to the intrinsic functions `MAXLOC` and `MINLOC`.
 - The new intrinsic function `MERGE_BITS` performs a bitwise merge using a mask.
 - The new intrinsic function `NORM2` calculates the L2 norm of an array.
 - The new intrinsic function `NUM_IMAGES` returns the number of images.
 - The new intrinsic function `PARITY` reduces an array with the `.NEQV.` operation.
 - The new intrinsic functions `POPCNT` and `POPPAR` return the number of 1 bits of an integer and its parity.
 - A `RADIX=` argument has been added to the intrinsic function `SELECTED_REAL_KIND`.
 - The new intrinsic functions `SHIFTA`, `SHIFTL` and `SHIFTR` perform shift operations.
 - The new intrinsic function `STORAGE_SIZE` returns the size of an array element in bits.
 - The new intrinsic function `THIS_IMAGE` returns the index of this image or cosubscripts for it.
- Intrinsic modules:

The functions `COMPILER_VERSION` and `COMPILER_OPTIONS` in the intrinsic module `ISO_FORTRAN_ENV` return information about the program translation phase. Named constants for selecting kind values have been added to the intrinsic module `ISO_FORTRAN_ENV`. A `RADIX=` argument has been added to the function `IEEE_SELECTED_REAL_KIND` in the intrinsic module `IEEE_ARITHMETIC`. The function `C_SIZEOF` in the intrinsic module `ISO_C_BINDING` returns the size of an array element in bytes. A `contiguous` array variable that is not interoperable but which has interoperable type and `kind type parameter` (if any), and a scalar character variable with length greater than one and `kind C_CHAR` in the intrinsic module `ISO_C_BINDING`, can be used as the argument of the function `C_LOC` in the intrinsic module `ISO_C_BINDING`, provided the variable has the `POINTER` or `TARGET` attribute.
 - Programs and procedures:

An empty `CONTAINS` section is allowed. An internal procedure can be used as an `actual argument` or `procedure pointer` target. `ALLOCATABLE` and `POINTER` attributes are used in generic resolution. Procedureness of a `dummy argument` is used in generic resolution. An `actual argument` with the `TARGET attribute` can correspond to a dummy pointer. A null pointer or unallocated allocatable can be used to denote an absent nonallocatable nonpointer optional argument. An impure `elemental` procedure processes array arguments in array element order. The `FUNCTION` and `SUBROUTINE` keywords can be omitted from the `END statement` for a module or internal subprogram. A line in the program is permitted to begin with a semicolon. An argument to a `pure procedure` can have default `INTENT` if it has the `VALUE attribute`. The name of an `external procedure` that has a `binding label` is a local identifier and not a global identifier. A procedure that is not a `procedure pointer` can be an `actual argument` that corresponds to a `procedure pointer dummy argument` with the `INTENT (IN) attribute`. An interface body for an `external procedure` that does not exist in a program can be used to specify an `explicit specific interface`. An `internal procedure` name can appear in a `procedure-stmt` in a `generic interface`

block.

C.3 Clause 7 notes

C.3.1 Selection of the approximation methods (7.4.3.2)

- 1 One can select the real approximation method for an entire program through the use of a module and the parameterized real type. This is accomplished by defining a named integer constant to have a particular kind type parameter value and using that [named constant](#) in all real, complex, and derived-type declarations. For example, the specification statements

```

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 \(16.9.183\)](#). 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.3.2 Type extension and component accessibility (7.5.2.2, 7.5.4)

- 1 The default accessibility of the components of an [extended type](#) can be specified in the type definition. The accessibility of its components can be specified individually. For example:

```

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

```

```

1      use types
2      type (my_type) :: x
3      ...
4      call another_sub( &
5          x%base_type,    &  !-- ok because base_type is a public subobject of x
6          x%base_type%k,  &  !-- ok because x%base_type is ok and has k as a
7                          !-- public component.
8          x%k,            &  !-- ok because it is shorthand for x%base_type%k
9          x%base_type%i,  &  !-- Invalid because i is private.
10         x%i)            !-- Invalid because it is shorthand for x%base_type%i
11     end subroutine sub

```

12 C.3.3 Generic type-bound procedures (7.5.5)

13 Example of a derived type with generic type-bound procedures:

- 14 1 The only difference between this example and the same thing rewritten to use generic [interface blocks](#) is
15 that with [type-bound procedures](#),

```

16     USE rational_numbers, ONLY: rational

```

17 does not block the [type-bound procedures](#); the user still gets access to the defined assignment and extended
18 operations.

```

19     MODULE rational_numbers
20     IMPLICIT NONE
21     PRIVATE
22     TYPE, PUBLIC :: rational
23     PRIVATE
24     INTEGER n,d
25     CONTAINS
26     ! ordinary type-bound procedure
27     PROCEDURE :: real => rat_to_real
28     ! specific type-bound procedures for generic support
29     PROCEDURE, PRIVATE :: rat_asgn_i, rat_plus_i, rat_plus_rat => rat_plus
30     PROCEDURE, PRIVATE, PASS(b) :: i_plus_rat
31     ! generic type-bound procedures
32     GENERIC :: ASSIGNMENT(=) => rat_asgn_i
33     GENERIC :: OPERATOR(+) => rat_plus_rat, rat_plus_i, i_plus_rat
34     END TYPE
35     CONTAINS
36     ELEMENTAL REAL FUNCTION rat_to_real(this) RESULT(r)
37     CLASS(rational), INTENT(IN) :: this
38     r = REAL(this%n)/this%d
39     END FUNCTION
40     ELEMENTAL SUBROUTINE rat_asgn_i(a,b)
41     CLASS(rational), INTENT(INOUT) :: a
42     INTEGER, INTENT(IN) :: b
43     a%n = b
44     a%d = 1
45     END SUBROUTINE
46     ELEMENTAL TYPE(rational) FUNCTION rat_plus_i(a,b) RESULT(r)
47     CLASS(rational), INTENT(IN) :: a
48     INTEGER, INTENT(IN) :: b
49     r%n = a%n + b*a%d
50     r%d = a%d
51     END FUNCTION
52     ELEMENTAL TYPE(rational) FUNCTION i_plus_rat(a,b) RESULT(r)
53     INTEGER, INTENT(IN) :: a
54     CLASS(rational), INTENT(IN) :: b

```

```

1      r%n = b%n + a*b%d
2      r%d = b%d
3  END FUNCTION
4  ELEMENTAL TYPE(rational) FUNCTION rat_plus(a,b) RESULT(r)
5      CLASS(rational), INTENT(IN) :: a,b
6      r%n = a%n*b%d + b%n*a%d
7      r%d = a%d*b%d
8  END FUNCTION
9  END

```

C.3.4 Abstract types (7.5.7.1)

- 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.

```

13  TYPE, ABSTRACT :: DRAWABLE_OBJECT
14      REAL, DIMENSION(3) :: RGB_COLOR = (/1.0,1.0,1.0/) ! White
15      REAL, DIMENSION(2) :: POSITION = (/0.0,0.0/) ! Centroid
16  CONTAINS
17      PROCEDURE(RENDER_X), PASS(OBJECT), DEFERRED :: RENDER
18  END TYPE DRAWABLE_OBJECT
19
20  ABSTRACT INTERFACE
21      SUBROUTINE RENDER_X(OBJECT, WINDOW)
22          IMPORT DRAWABLE_OBJECT, X_WINDOW
23          CLASS(DRAWABLE_OBJECT), INTENT(IN) :: OBJECT
24          CLASS(X_WINDOW), INTENT(INOUT) :: WINDOW
25      END SUBROUTINE RENDER_X
26  END INTERFACE
27
28  ...
29
30  TYPE, EXTENDS(DRAWABLE_OBJECT) :: DRAWABLE_TRIANGLE ! Not ABSTRACT
31      REAL, DIMENSION(2,3) :: VERTICES ! In relation to centroid
32  CONTAINS
33      PROCEDURE, PASS(OBJECT) :: RENDER=>RENDER_TRIANGLE_X
34  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):

```

37  SUBROUTINE RENDER_TRIANGLE_X(OBJECT, WINDOW)
38      CLASS(DRAWABLE_TRIANGLE), INTENT(IN) :: OBJECT
39      CLASS(X_WINDOW), INTENT(INOUT) :: WINDOW
40      ...
41  END SUBROUTINE RENDER_TRIANGLE_X

```

C.3.5 Structure constructors and generic names (7.5.10)

- 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:

```

45  MODULE mytype_module
46      TYPE mytype
47          PRIVATE
48          COMPLEX value
49          LOGICAL exact

```

```

1      END TYPE
2      INTERFACE mytype
3          MODULE PROCEDURE int_to_mytype
4      END INTERFACE
5      ! Operator definitions etc.
6      ...
7  CONTAINS
8      TYPE(mytype) FUNCTION int_to_mytype(i)
9          INTEGER, INTENT(IN) :: i
10         int_to_mytype%value = i
11         int_to_mytype%exact = .TRUE.
12     END FUNCTION
13     ! Procedures to support operators etc.
14     ...
15 END
16
17 PROGRAM example
18     USE mytype_module
19     TYPE(mytype) x
20     x = mytype(17)
21 END

```

22 2 The type name can still be used as a generic name if the type has type parameters. For example:

```

23     MODULE m
24         TYPE t(kind)
25             INTEGER, KIND :: kind
26             COMPLEX(kind) value
27         END TYPE
28         INTEGER, PARAMETER :: single = KIND(0.0), double = KIND(0d0)
29         INTERFACE t
30             MODULE PROCEDURE real_to_t1, dble_to_t2, int_to_t1, int_to_t2
31         END INTERFACE
32         ...
33     CONTAINS
34         TYPE(t(single)) FUNCTION real_to_t1(x)
35             REAL(single) x
36             real_to_t1%value = x
37         END FUNCTION
38         TYPE(t(double)) FUNCTION dble_to_t2(x)
39             REAL(double) x
40             dble_to_t2%value = x
41         END FUNCTION
42         TYPE(t(single)) FUNCTION int_to_t1(x,mold)
43             INTEGER x
44             TYPE(t(single)) mold
45             int_to_t1%value = x
46         END FUNCTION
47         TYPE(t(double)) FUNCTION int_to_t2(x,mold)
48             INTEGER x
49             TYPE(t(double)) mold
50             int_to_t2%value = x
51         END FUNCTION
52         ...
53     END
54
55     PROGRAM example
56         USE m
57         TYPE(t(single)) x
58         TYPE(t(double)) y
59         x = t(1.5)                                ! References real_to_t1

```

```

1      x = t(17,mold=x)           ! References int_to_t1
2      y = t(1.5d0)              ! References dble_to_t2
3      y = t(42,mold=y)          ! References int_to_t2
4      y = t(kind(0d0)) ((0,1))  ! Uses the structure constructor for type t
5      END

```

6 C.3.6 Final subroutines (7.5.6, 7.5.6.2, 7.5.6.3, 7.5.6.4)

7 Example of a parameterized derived type with final subroutines:

```

8      MODULE m
9      TYPE t(k)
10         INTEGER, KIND :: k
11         REAL(k), POINTER :: vector(:) => NULL()
12     CONTAINS
13         FINAL :: finalize_t1s, finalize_t1v, finalize_t2e
14     END TYPE
15 CONTAINS
16     SUBROUTINE finalize_t1s(x)
17         TYPE(t(KIND(0.0))) x
18         IF (ASSOCIATED(x%vector)) DEALLOCATE(x%vector)
19     END SUBROUTINE
20     SUBROUTINE finalize_t1v(x)
21         TYPE(t(KIND(0.0))) x(:)
22         DO i=LBOUND(x,1),UBOUND(x,1)
23             IF (ASSOCIATED(x(i)%vector)) DEALLOCATE(x(i)%vector)
24         END DO
25     END SUBROUTINE
26     ELEMENTAL SUBROUTINE finalize_t2e(x)
27         TYPE(t(KIND(0.0d0))),INTENT(INOUT) :: x
28         IF (ASSOCIATED(x%vector)) DEALLOCATE(x%vector)
29     END SUBROUTINE
30 END MODULE
31
32 SUBROUTINE example(n)
33     USE m
34     TYPE(t(KIND(0.0))) a,b(10),c(n,2)
35     TYPE(t(KIND(0.0d0))) d(n,n)
36     ...
37     ! Returning from this subroutine will effectively do
38     !     CALL finalize_t1s(a)
39     !     CALL finalize_t1v(b)
40     !     CALL finalize_t2e(d)
41     ! No final subroutine will be called for variable C because the user
42     ! omitted to define a suitable specific procedure for it.
43 END SUBROUTINE

```

44 Example of extended types with final subroutines:

```

45     MODULE m
46     TYPE t1
47         REAL a,b
48     END TYPE
49     TYPE,EXTENDS(t1) :: t2
50         REAL,POINTER :: c(:),d(:)
51     CONTAINS
52         FINAL :: t2f
53     END TYPE
54     TYPE,EXTENDS(t2) :: t3

```

```

1      REAL, POINTER :: e
2      CONTAINS
3      FINAL :: t3f
4      END TYPE
5      ...
6      CONTAINS
7      SUBROUTINE t2f(x) ! Finalizer for TYPE(t2)'s extra components
8      TYPE(t2) :: x
9      IF (ASSOCIATED(x%c)) DEALLOCATE(x%c)
10     IF (ASSOCIATED(x%d)) DEALLOCATE(x%d)
11     END SUBROUTINE
12     SUBROUTINE t3f(y) ! Finalizer for TYPE(t3)'s extra components
13     TYPE(t3) :: y
14     IF (ASSOCIATED(y%e)) DEALLOCATE(y%e)
15     END SUBROUTINE
16 END MODULE
17
18 SUBROUTINE example
19     USE m
20     TYPE(t1) x1
21     TYPE(t2) x2
22     TYPE(t3) x3
23     ...
24     ! Returning from this subroutine will effectively do
25     !     ! Nothing to x1; it is not finalizable
26     !     CALL t2f(x2)
27     !     CALL t3f(x3)
28     !     CALL t2f(x3%t2)
29 END SUBROUTINE

```

30 C.4 Clause 8 notes: The VOLATILE attribute (8.5.20)

- 31 1 The following example shows the use of a variable with the **VOLATILE attribute** to communicate with an
32 asynchronous process, in this case the operating system. The program detects a user keystroke on the
33 terminal and reacts at a convenient point in its processing.
- 34 2 The **VOLATILE attribute** is necessary to prevent an optimizing compiler from storing the communication
35 variable in a register or from doing flow analysis and deciding that the **EXIT statement** can never be ex-
36 ecuted.

```

37 SUBROUTINE TERMINATE_ITERATIONS
38     LOGICAL, VOLATILE :: USER_HIT_ANY_KEY
39
40     ! Have the OS start to look for a user keystroke and set the variable
41     ! "USER_HIT_ANY_KEY" to TRUE as soon as it detects a keystroke.
42     ! This call is operating system dependent.
43
44     CALL OS_BEGIN_DETECT_USER_KEYSTROKE( USER_HIT_ANY_KEY )
45     USER_HIT_ANY_KEY = .FALSE. ! This will ignore any recent keystrokes.
46     PRINT *, " Hit any key to terminate iterations!"
47
48     DO I = 1, 100
49         ... Compute a value for R.
50         PRINT *, I, R
51         IF (USER_HIT_ANY_KEY) EXIT
52     ENDDO
53
54     ! Have the OS stop looking for user keystrokes.

```

```

1      CALL OS_STOP_DETECT_USER_KEYSTROKE
2      END SUBROUTINE TERMINATE_ITERATIONS

```

C.5 Clause 9 notes

C.5.1 Structure components (9.4.2)

- 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,

```

7      TYPE ID_NUMBERS
8          INTEGER SSN
9          INTEGER EMPLOYEE_NUMBER
10     END TYPE ID_NUMBERS
11
12     TYPE PERSON_ID
13         CHARACTER (LEN=30) LAST_NAME
14         CHARACTER (LEN=1) MIDDLE_INITIAL
15         CHARACTER (LEN=30) FIRST_NAME
16         TYPE (ID_NUMBERS) NUMBER
17     END TYPE PERSON_ID
18
19     TYPE PERSON
20         INTEGER AGE
21         TYPE (PERSON_ID) ID
22     END TYPE PERSON
23
24     TYPE (PERSON) GEORGE, MARY
25
26     PRINT *, GEORGE % AGE           ! Print the AGE component
27     PRINT *, MARY % ID % LAST_NAME ! Print LAST_NAME of MARY
28     PRINT *, MARY % ID % NUMBER % SSN ! Print SSN of MARY
29     PRINT *, GEORGE % ID % NUMBER ! Print SSN and EMPLOYEE_NUMBER of GEORGE

```

- 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.

```

33     TYPE LARGE
34         INTEGER ELT (10)
35         INTEGER VAL
36     END TYPE LARGE
37
38     TYPE (LARGE) A (5)           ! 5 element array, each of whose elements
39                                 ! includes a 10 element array ELT and
40                                 ! a scalar VAL.
41     PRINT *, A (1)               ! Prints 10 element array ELT and scalar VAL.
42     PRINT *, A (1) % ELT (3)    ! Prints scalar element 3
43                                 ! of array element 1 of A.
44     PRINT *, A (2:4) % VAL      ! Prints scalar VAL for array elements
45                                 ! 2 to 4 of A.

```

- 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:


```

1      TYPE POINT                ! A base type
2      REAL :: X, Y
3  END TYPE POINT
4  TYPE, EXTENDS(POINT) :: COLOR_POINT ! An extension of TYPE(POINT)
5      ! Components X and Y, and component name POINT, inherited from parent
6      INTEGER :: COLOR
7  END TYPE COLOR_POINT
8
9  TYPE(POINT), PARAMETER :: PV = POINT(1.0, 2.0)
10 TYPE(COLOR_POINT) :: CPV = COLOR_POINT(POINT=PV, COLOR=3)
11
12 PRINT *, CPV%POINT                ! Prints 1.0 and 2.0
13 PRINT *, CPV%POINT%X, CPV%POINT%Y ! And this does, too
14 PRINT *, CPV%X, CPV%Y             ! And this does, too

```

C.5.2 Allocation with dynamic type (9.7.1)

- 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.

```

21 TYPE :: LIST ! A list of anything
22 TYPE(LIST), POINTER :: NEXT => NULL()
23 CLASS(*), ALLOCATABLE :: ITEM
24 END TYPE LIST
25 ...
26 TYPE(LIST), POINTER :: IN_LIST, LIST_COPY => NULL()
27 TYPE(LIST), POINTER :: IN_WALK, NEW_TAIL
28 ! Copy IN_LIST to LIST_COPY
29 IF (ASSOCIATED(IN_LIST)) THEN
30     IN_WALK => IN_LIST
31     ALLOCATE(LIST_COPY)
32     NEW_TAIL => LIST_COPY
33     DO
34         ALLOCATE(NEW_TAIL%ITEM, SOURCE=IN_WALK%ITEM)
35         IN_WALK => IN_WALK%NEXT
36         IF (.NOT. ASSOCIATED(IN_WALK)) EXIT
37         ALLOCATE(NEW_TAIL%NEXT)
38         NEW_TAIL => NEW_TAIL%NEXT
39     END DO
40 END IF

```

C.6 Clause 10 notes

C.6.1 Evaluation of function references (10.1.7)

- 1 If more than one function reference appears in a statement, they can be executed in any order (subject to a function result being evaluated after the evaluation of its arguments) and their values cannot depend on the order of execution. This lack of dependence on order of evaluation enables parallel execution of the function references.

1 C.6.2 Pointers in expressions (10.1.9.2)

2 1 A **data pointer** is considered to be like any other variable when it is used as a primary in an expression. If
 3 a pointer is used as an operand to an operator that expects a value, the pointer will automatically deliver
 4 the value stored in the space described by the pointer, that is, the value of the target object associated with
 5 the pointer.

6 C.6.3 Pointers in variable definition contexts (10.2.1.3, 19.6.7)

7 1 The appearance of a **data pointer** in a context that requires its value is a reference to its target. Similarly,
 8 where a pointer appears in a variable definition context the variable that is defined is the target of the
 9 pointer.

10 2 Executing the program fragment

```
11      REAL, POINTER :: A
12      REAL, TARGET :: B = 10.0
13      A => B
14      A = 42.0
15      PRINT ' (F4.1) ', B
```

16 produces “42.0” as output.

17 C.7 Clause 11 notes

18 C.7.1 The SELECT CASE construct (11.1.9)

19 1 At most one case block is selected for execution within a **SELECT CASE construct**, and there is no fall-
 20 through from one block into another block within a **SELECT CASE construct**. Thus there is no requirement
 21 for the user to exit explicitly from a block.

22 C.7.2 Loop control (11.1.7)

23 1 Fortran provides several forms of loop control:

- 24 (1) With an iteration count and a DO variable. This is the classic Fortran DO loop.
- 25 (2) Test a logical condition before each execution of the loop (DO WHILE).
- 26 (3) DO “forever”.

27 C.7.3 Examples of DO constructs (11.1.7)

28 1 The following are all valid examples of **DO constructs**.

29 Example 1:

```
30      SUM = 0.0
31      READ (IUN) N
32      OUTER: DO I = 1, N                ! A DO with a construct name
33          READ (IUN) IQUAL, M, ARRAY (1:M)
34          IF (IQUAL < IQUAL_MIN) CYCLE OUTER ! Skip inner loop
35          INNER: DO 40 I = 1, M          ! A DO with a label and a name
36              CALL CALCULATE (ARRAY (I), RESULT)
37              IF (RESULT < 0.0) CYCLE
38              SUM = SUM + RESULT
```

```

1           IF (SUM > SUM_MAX) EXIT OUTER
2   40      END DO INNER
3          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.

10 Example 2:

```

11      N = 0
12      DO 50, I = 1, 10
13          J = I
14          DO K = 1, 5
15              L = K
16              N = N + 1 ! This statement executes 50 times
17          END DO      ! Nonlabeled DO inside a labeled DO
18 50 CONTINUE

```

- 3 After execution of the above program fragment, I = 11, J = 10, K = 6, L = 5, and N = 50.

20 Example 3:

```

21      N = 0
22      DO I = 1, 10
23          J = I
24          DO 60, K = 5, 1 ! This inner loop is never executed
25              L = K
26              N = N + 1
27 60 CONTINUE            ! Labeled DO inside a nonlabeled DO
28      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.

31 C.7.4 Examples of invalid DO constructs (11.1.7)

- 1 The following are all examples of invalid skeleton **DO** constructs:

33 Example 1:

```

34      DO I = 1, 10
35      ...
36      END DO LOOP ! No matching construct name

```

37 Example 2:

```

38      LOOP: DO 1000 I = 1, 10 ! No matching construct name
39      ...
40      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.7.5 Simple example using events

- 1 A tree is a graph in which every node except one has a single “parent” node to which it is connected by an edge. The node without a parent is the “root” of the tree. The nodes that have a particular node as their parent are the “children” of that node. The root is at level 1, its children are at level 2, and so on.
- 2 A multifrontal code to solve a sparse set of linear equations involves a tree. Work at a node can start after all of its children’s work is complete and their data have been passed to it.
- 3 Here we assume that each node has been assigned to an image. Each image has a list of its nodes and these are ordered in decreasing tree level (all those at level L preceding those at level $L - 1$). For each node, array elements hold the number of children, details about the parent, and an event variable. This allows the processing to proceed asynchronously subject to the rule that a parent has to wait for all its children.

Outline of example code:

```

PROGRAM TREE
  USE, INTRINSIC :: ISO_FORTRAN_ENV
  INTEGER, ALLOCATABLE :: NODE (:) ! Tree nodes that this image handles.
  INTEGER, ALLOCATABLE :: NC (:)    ! NODE(I) has NC(I) children.
  INTEGER, ALLOCATABLE :: PARENT (:), SUB (:)
  ! The parent of NODE (I) is NODE (SUB (I)) [PARENT (I)].
  TYPE (EVENT_TYPE), ALLOCATABLE :: DONE (:) [:]
  INTEGER :: I, J, STATUS
  ! Set up the tree, including allocation of all arrays.
  DO I = 1, SIZE (NODE)
    ! Wait for children to complete
    IF (NC (I) > 0) THEN
      EVENT WAIT (DONE (I), UNTIL_COUNT=NC (I), STAT=STATUS)

```

```

1         IF (STATUS/=0) EXIT
2     END IF
3
4     ! Process node, using data from children.
5     IF (PARENT (I)>0) THEN
6         ! Node is not the root.
7         ! Place result on image PARENT (I) for node NODE (SUB) [PARENT (I)]
8         ! Tell PARENT (I) that this has been done.
9         EVENT POST (DONE (SUB (I)) [PARENT (I)], STAT=STATUS)
10        IF (STATUS/=0) EXIT
11    END IF
12 END DO
13 END PROGRAM TREE

```

C.7.6 Example using three teams

- 1 The following example illustrates the structure of a routine that will compute fluxes based on surface properties over land, sea, and ice, each in a different team. Each image will deal with areas containing exactly one of the three surface types.

```

18 SUBROUTINE COMPUTE_FLUXES (FLUX_MOM, FLUX_SENS, FLUX_LAT)
19 USE, INTRINSIC :: ISO_FORTRAN_ENV, ONLY: TEAM_TYPE
20 REAL, INTENT (OUT) :: FLUX_MOM (:,:), FLUX_SENS (:,:), FLUX_LAT (:,:)
21 INTEGER, PARAMETER :: LAND = 1, SEA = 2, ICE = 3
22 CHARACTER (LEN=10) :: SURFACE_TYPE
23 INTEGER :: MY_SURFACE_TYPE, N_IMAGE
24 TYPE (TEAM_TYPE) :: TEAM_SURFACE_TYPE
25
26 CALL GET_SURFACE_TYPE(THIS_IMAGE (), SURFACE_TYPE)
27 SELECT CASE (SURFACE_TYPE)
28 CASE ("LAND")
29     MY_SURFACE_TYPE = LAND
30 CASE ("SEA")
31     MY_SURFACE_TYPE = SEA
32 CASE ("ICE")
33     MY_SURFACE_TYPE = ICE
34 CASE DEFAULT
35     ERROR STOP
36 END SELECT
37 FORM TEAM (MY_SURFACE_TYPE, TEAM_SURFACE_TYPE)
38
39 CHANGE TEAM (TEAM_SURFACE_TYPE)
40 SELECT CASE (TEAM_NUMBER ( ))
41 CASE (LAND) ! Compute fluxes over land surface
42     CALL COMPUTE_FLUXES_LAND (FLUX_MOM, FLUX_SENS, FLUX_LAT)
43 CASE (SEA) ! Compute fluxes over sea surface
44     CALL COMPUTE_FLUXES_SEA (FLUX_MOM, FLUX_SENS, FLUX_LAT)
45 CASE (ICE) ! Compute fluxes over ice surface
46     CALL COMPUTE_FLUXES_ICE (FLUX_MOM, FLUX_SENS, FLUX_LAT)
47 CASE DEFAULT
48     ERROR STOP
49 END SELECT
50 END TEAM
51 END SUBROUTINE COMPUTE_FLUXES

```

C.7.7 Accessing coarrays in sibling teams

- 1 The following program illustrates subdividing a 4×4 grid into 2×2 teams, and the denotation of sibling teams.

```

1      PROGRAM DEMO
2          ! Initial team : 16 images. Algorithm design is a 4 by 4 grid.
3          ! Desire 4 teams, for the upper left (UL), upper right (UR),
4          !             lower left (LL), lower right (LR)
5          USE, INTRINSIC :: ISO_FORTRAN_ENV, ONLY: TEAM_TYPE
6          TYPE (TEAM_TYPE) :: T
7          INTEGER, PARAMETER :: UL=11, UR=22, LL=33, LR=44
8          REAL :: A(10,10)[4,*]
9          INTEGER :: MYPE, TEAMNUM, NEWPE
10         TYPE TRANS_T
11             INTEGER :: NEW_TEAM (16), NEW_INDEX (16)
12         END TYPE
13         TYPE (TRANS_T) :: TX
14         TX = TRANS_T ([UL, UL, LL, LL, UL, UL, LL, LL, UR, UR, LR, LR, UR, UR, LR, LR], &
15             [1, 2, 1, 2, 3, 4, 3, 4, 1, 2, 1, 2, 3, 4, 3, 4])
16
17         MYPE = THIS_IMAGE ()
18         FORM TEAM (TX%NEW_TEAM(MYPE), T, NEW_INDEX=TX%NEW_INDEX(MYPE))
19
20         A = 3.14
21
22         CHANGE TEAM (T, B[2,*] => A)
23         ! Inside change team, image pattern for B is a 2 by 2 grid.
24         B (5, 5) = B (1, 1)[2, 1]
25
26         ! Outside the team addressing:
27
28         NEWPE = THIS_IMAGE ()
29         SELECT CASE (TEAM_NUMBER ())
30         CASE (UL)
31             IF (NEWPE==3) THEN
32                 ! Right column of UL gets left column of UR.
33                 B (:, 10) = B (:, 1)[1, 1, TEAM_NUMBER=UR]
34             ELSE IF (NEWPE==4) THEN
35                 B (:, 10) = B (:, 1)[2, 1, TEAM_NUMBER=UR]
36             END IF
37         CASE (LL)
38             ! Similar to complete column exchange across middle of the original grid.
39             ...
40         END SELECT
41     END TEAM
42 END PROGRAM DEMO

```

43 C.7.8 Example involving failed images

- 44 1 Parallel algorithms often use work sharing schemes based on a specific mapping between image indices and
45 global data addressing. To allow such programs to continue when one or more images fail, spare images
46 can be used to re-establish execution of the algorithm with the failed images replaced by spare images,
47 while retaining the previous image mapping for nonfailed images.
- 48 2 The following example illustrates how this might be done. In this example, failure cannot be tolerated for
49 image one in the initial team.

```

50 PROGRAM possibly_recoverable_simulation
51     USE, INTRINSIC :: ISO_FORTRAN_ENV, ONLY: TEAM_TYPE, STAT_FAILED_IMAGE
52     IMPLICIT NONE
53     INTEGER :: images_spare ! Number of spare images.
54     INTEGER :: images_used ! Number of images used.
55     INTEGER :: j, k ! Temporaries

```

```

1      INTEGER :: status          ! STAT= value
2      INTEGER :: team_number [*] ! 1 if in working team; 2 otherwise.
3      INTEGER :: local_index [*] ! Index of the image in the team.
4      TYPE (TEAM_TYPE) :: simulation_team
5      LOGICAL :: done [*]       ! True if computation finished on the image.
6
7      ! Keep 1% spare images if we have a lot, just 1 if 10-199 images,
8      !                               0 if <10.
9      images_spare = MAX(NUM_IMAGES()/100,0,MIN(NUM_IMAGES()-9,1))
10     images_used = NUM_IMAGES () - images_spare
11     SYNC ALL (STAT=status)
12
13     outer : DO
14         IF (status/=0 .AND. status/=STAT_FAILED_IMAGE) EXIT outer
15         IF (IMAGE_STATUS (1) == STAT_FAILED_IMAGE) ERROR STOP "cannot recover"
16         IF (THIS_IMAGE () == 1) THEN
17             j = 0
18             DO k = 1, NUM_IMAGES ()
19                 IF (IMAGE_STATUS (k) == 0) THEN
20                     j = j+1
21                     IF (j<=images_used) THEN
22                         local_index[k] = j
23                         team_number [k] = 1
24                     ELSE
25                         local_index[k] = j - images_used
26                         team_number [k] = 2
27                     END IF
28                 END IF
29             END DO
30             IF (j<images_used) ERROR STOP "cannot recover"
31         END IF
32         SYNC ALL (STAT = status)
33         IF (status/=0 .AND. status/=STAT_FAILED_IMAGE) EXIT outer
34         ! Set up a simulation team of constant size.
35         ! Team 2 is the set of spares, so does not participate.
36         FORM TEAM (team_number, simulation_team, NEW_INDEX=local_index, STAT=status)
37         IF (status/=0 .AND. status/=STAT_FAILED_IMAGE) EXIT outer
38
39         simulation : CHANGE TEAM (simulation_team, STAT=status)
40         IF (status == STAT_FAILED_IMAGE) EXIT simulation
41         IF (team_number == 1) THEN
42             ! Each working image reads checkpoint data for itself if available.
43             iter : DO
44                 CALL simulation_procedure (status, done)
45                 ! The simulation_procedure:
46                 ! - sets up and performs some part of the simulation;
47                 ! - stores checkpoint data for all images from time to time;
48                 ! - sets status from its internal synchronizations so it has
49                 !   the value STAT_FAILED_IMAGE on all active images of the
50                 !   team if any image of the team has failed;
51                 ! - sets done to .TRUE. when the simulation has completed.
52                 IF (status == STAT_FAILED_IMAGE) THEN
53                     EXIT simulation
54                 ELSE IF (done) THEN
55                     EXIT iter
56                 END IF
57             END DO iter
58         END IF
59         END TEAM (STAT=status) simulation
60         IF (status/=0 .AND. status/=STAT_FAILED_IMAGE) EXIT outer
61

```



```

1      SYNC ALL (STAT=status)
2      IF (team_number == 2) done = done[1]
3      IF (done) EXIT outer
4  END DO outer
5      IF (status/=0 .AND. status/=STAT_FAILED_IMAGE) &
6          PRINT *, 'Unexpected failure', status
7  END PROGRAM possibly_recoverable_simulation

```

- 3 Supporting fault-tolerant execution imposes obligations on library writers who use the parallel language facilities. Every synchronization statement, allocation or deallocation of coarrays, or invocation of a [collective procedure](#) will need to be prepared to handle error conditions, and implicit deallocation of coarrays will need to be avoided. Also, coarray module variables that are allocated inside the [team](#) execution context are not persistent.

C.7.9 EVENT_QUERY example that tolerates image failure

- 1 This example is an adaptation of the later [EVENT_QUERY](#) example of [C.12.2](#) to make it able to execute in the presence of the failure of one or more of the worker images. The function `create_work_item` now accepts an integer argument to indicate which work item is required. It is assumed that the work items are indexed 1, 2, and so on. It is also assumed that if an image fails while processing a work item, that work item can subsequently be processed by another image.

```

19  PROGRAM work_share
20      USE, INTRINSIC :: ISO_FORTRAN_ENV, ONLY: EVENT_TYPE
21      USE :: mod_work, ONLY: & ! Module that creates work items
22          work, & ! Type for holding a work item
23          create_work_item, & ! Function that creates work item
24          process_item, & ! Function that processes an item
25          work_done ! Logical function that returns true
26                  ! if all work done
27
28      TYPE :: worker_type
29          TYPE (EVENT_TYPE), ALLOCATABLE :: free (:)
30  END TYPE
31      TYPE (EVENT_TYPE) :: submit [*] ! Whether work ready for a worker
32      TYPE (worker_type) :: worker [*] ! Whether worker is free
33      TYPE (work) :: work_item [*] ! Holds the data for a work item
34      INTEGER :: count, i, k, kk, nbusy [*, np, status
35      INTEGER, ALLOCATABLE :: working (:) ! Items being worked on
36      INTEGER, ALLOCATABLE :: pending (:) ! Items pending after image failure
37
38      IF (THIS_IMAGE () == 1) THEN
39          ! Get started
40          ALLOCATE (worker%free (2:NUM_IMAGES ()))
41          ALLOCATE (working (2: NUM_IMAGES ()), pending(NUM_IMAGES ()-1))
42          nbusy = 0 ! This holds the number of workers working
43          k = 1 ! Index of next work item
44          np = 0 ! Number of work items in array pending
45          DO i = 2, NUM_IMAGES () ! Start the workers working
46              IF (work_done ()) EXIT
47              working (i) = 0
48              IF (IMAGE_STATUS (i) == STAT_FAILED_IMAGE) CYCLE
49              work_item [i] = create_work_item (k)
50              working (i) = k
51              k = k + 1
52              nbusy = nbusy + 1
53              EVENT POST (submit [i], STAT=status)
54          END DO

```

```

1      ! Main work distribution loop
2  main : DO
3      image : DO i = 2, NUM_IMAGES ()
4          IF (IMAGE_STATUS (i) == STAT_FAILED_IMAGE) THEN
5              IF (working (i)>0) THEN ! It failed while working
6                  np = np + 1
7                  pending (np) = working (i)
8                  working (i) = 0
9              END IF
10             CYCLE image
11         END IF
12         CALL EVENT_QUERY (worker%free (i), count)
13         IF (count == 0) CYCLE image ! Worker is not free
14         EVENT WAIT (worker%free (i))
15         nbusy = nbusy - 1
16         IF (np>0) THEN
17             kk = pending (np)
18             np = np - 1
19         ELSE
20             IF (work_done ()) CYCLE image
21             kk = k
22             k = k + 1
23         END IF
24         nbusy = nbusy + 1
25         working (i) = kk
26         work_item [i] = create_work_item (kk)
27         EVENT POST (submit [i], STAT=status)
28         ! If image i has failed, the failure will be handled on
29         ! the next iteration of the main loop.
30     END DO image
31     IF ( nbusy==0 ) THEN ! All done. Exit on all images.
32         DO i = 2, NUM_IMAGES ()
33             EVENT POST (submit [i], STAT=status)
34             IF (status == STAT_FAILED_IMAGE) CYCLE
35         END DO
36         EXIT main
37     END IF
38 END DO main
39 ELSE
40     ! Work processing loop
41     worker : DO
42         EVENT WAIT (submit)
43         IF (nbusy [1] == 0) EXIT worker
44         CALL process_item(work_item)
45         EVENT POST (worker[1]%free (THIS_IMAGE ()))
46     END DO worker
47 END IF
48 END PROGRAM work_share

```

C.8 Clause 12 notes

C.8.1 External files (12.3)

C.8.1.1 File cataloging

- 1 This document accommodates, but does not require, file cataloging. To do this, several concepts are intro-
- 53 duced.

C.8.1.2 File existence (12.3.2)

Totally independent of the connection state is the property of existence, this being a file property. The processor “knows” of a set of files that exist at a given time for a given program. This set would include tapes ready to read, files in a catalog, a keyboard, a printer, etc. The set might exclude files inaccessible to the program because of security, because they are already in use by another program, etc. This document does not specify which files exist, hence wide latitude is available to a processor to implement security, locks, privilege techniques, etc. Existence is a convenient concept to designate all of the files that a program can potentially process.

All four combinations of connection and existence can occur:

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

Means are provided to create, delete, connect, and disconnect files.

C.8.1.3 File access (12.3.3)

This document does not address problems of security, protection, locking, and many other concepts that might be part of the concept of “right of access”. Such concepts are considered to be in the province of an operating system.

The **OPEN** and **INQUIRE** statements can be extended naturally to consider these things.

Possible access methods for a file are: sequential, stream and direct. The processor might implement three different types of files, each with its own access method. It might instead implement one type of file with three different access methods.

Direct access to files is of a simple and commonly available type, that is, fixed-length records. The key is a positive integer.

C.8.1.4 File connection (12.5)

Before any input/output can be performed on a file, it needs to be connected to a **unit**. The **unit** then serves as a designator for that file as long as it is connected. To be connected does not imply that “buffers” have or have not been allocated, that “file-control tables” have or have not been filled, or that any other method of implementation has been used. Connection means that (barring some other fault) a **READ** or **WRITE** statement can be executed on the **unit**, hence on the file. Without a connection, a **READ** or **WRITE** statement cannot be executed.

C.8.1.5 File names (12.5.6.10)

A file can have a name. The form of a file name is not specified. If a system does not have some form of cataloging or tape labeling for at least some of its files, all file names disappear at the **termination of execution**. This is a valid implementation. Nowhere does this document require names to survive for any period of time longer than the execution time span of a program. Therefore, this document does not impose

cataloging as a prerequisite. The naming feature is intended to enable use of a cataloging system where one exists.

C.8.2 Nonadvancing input/output (12.3.4.2)

Data transfer statements affect the positioning of an **external file**. In FORTRAN 77, if no error or end-of-file condition exists, the file is positioned after the record just read or written and that record becomes the preceding record. This document contains the **ADVANCE= specifier** in a **data transfer statement** that provides the capability of maintaining a position within the current record from one formatted **data transfer statement** to the next **data transfer statement**. The value NO provides this capability. The value YES positions the file after the record just read or written. The default is YES.

The tab edit descriptor and the slash are still appropriate for use with this type of record access but the tab cannot reposition before the left tab limit.

A **BACKSPACE** of a file that is positioned within a record causes the specified unit to be positioned before the current record.

If the next input/output operation on a file after a nonadvancing write is a **rewind**, **backspace**, **end file** or **close** operation, the file is positioned implicitly after the current record before an ENDFILE record is written to the file, that is, a **REWIND**, **BACKSPACE**, or **ENDFILE** statement following a nonadvancing **WRITE statement** causes the file to be positioned at the end of the current output record before the endfile record is written to the file.

This document provides a **SIZE= specifier** to be used with formatted **data transfer statements**. The variable in the **SIZE= specifier** is assigned the count of the number of characters that make up the sequence of values read by the data edit descriptors in the **input statement**. The count is especially helpful if there is only one **effective item** in the input list because it is the number of characters that appeared for the item.

The **EOR= specifier** is provided to indicate when an EOR condition is encountered during nonadvancing **input**. The EOR condition is not an error condition. If this specifier appears, an **effective item** that requires more characters than the record contained is padded with blanks if **PAD= 'YES'** is in effect. This means that input of the **effective item** completed successfully. The file is positioned after the current record. If the **IO-STAT= specifier** appears, the specified variable is defined with the value of the **named constant IOSTAT_EOR** from the intrinsic module **ISO_FORTRAN_ENV** and the **data transfer statement** is terminated. Program execution continues with the statement specified in the **EOR= specifier**. The **EOR= specifier** gives the capability of taking control of execution when the EOR condition is encountered. The *do-variables* in *io-implied-dos* retain their last defined value and any remaining items in the *input-item-list* retain their definition status when an EOR condition occurs. If the **SIZE= specifier** appears, the specified variable is assigned the number of characters read with the data edit descriptors during the **READ statement**.

For nonadvancing input, the processor is not required to read partial records. The processor could read the entire record into an internal buffer and make successive portions of the record available to successive **input statements**.

In an implementation of nonadvancing input/output in which a nonadvancing write to a terminal device causes immediate display of the output, such a write can be used as a mechanism to output a prompt. In this case, the statement

```
WRITE (*, FMT='(A) ', ADVANCE='NO') 'CONTINUE?(Y/N): '
```

would result in the prompt

1 CONTINUE? (Y/N) :

2 being displayed with no subsequent line feed.

3 9 The response, which might be read by a statement of the form

4 READ (*, FMT='(A) ') ANSWER

5 can then be entered on the same line as the prompt as in

6 CONTINUE? (Y/N) : Y

7 10 This document does not require that an implementation of nonadvancing input/output operate in this man-
8 ner. For example, an implementation of nonadvancing output in which the display of the output is deferred
9 until the current record is complete is also standard-conforming. Such an implementation will not, how-
10 ever, allow a prompting mechanism of this kind to operate.

11 C.8.3 OPEN statement (12.5.6)

12 1 A file can become connected to a [unit](#) either by preconnection or by execution of an [OPEN statement](#). Pre-
13 connection is performed prior to the beginning of execution of a program by means external to Fortran.
14 For example, it could be done by job control action or by processor-established defaults. Execution of an
15 [OPEN statement](#) is not required in order to access preconnected files ([12.5.5](#)).

16 2 The [OPEN statement](#) provides a means to access existing files that are not preconnected. An [OPEN state-](#)
17 [ment](#) can be used in either of two ways: with a file name (open-by-name) and without a file name (open-
18 by-unit). A [unit](#) is given in either case. Open-by-name connects the specified file to the specified [unit](#).
19 Open-by-unit connects a processor-dependent default file to the specified [unit](#). (The default file might or
20 might not have a name.)

21 3 Therefore, there are three ways a file can become connected and hence processed: preconnection, open-
22 by-name, and open-by-unit. Once a file is connected, there is no means in standard Fortran to determine
23 how it became connected.

24 4 An [OPEN statement](#) can also be used to create a new file. In fact, any of the foregoing three connection
25 methods can be performed on a file that does not exist. When a [unit](#) is preconnected, writing the first
26 record creates the file. With the other two methods, execution of the [OPEN statement](#) creates the file.

27 5 When an [OPEN statement](#) is executed, the [unit](#) specified in the [OPEN statement](#) might or might not already
28 be connected to a file. If it is already connected to a file (either through preconnection or by prior execution
29 of an [OPEN statement](#)), then omitting the [FILE= specifier](#) in the [OPEN statement](#) implies that the file is
30 to remain connected to the [unit](#). Such an [OPEN statement](#) can be used to change the values of the blank
31 interpretation mode, decimal edit mode, pad mode, input/output rounding mode, delimiter mode, and sign
32 mode.

33 6 If the value of the [ACTION= specifier](#) is WRITE, then a [READ statement](#) cannot refer to the connection.
34 [ACTION = 'WRITE'](#) does not restrict positioning by a [BACKSPACE statement](#) or positioning specified by
35 the [POSITION= specifier](#) with the value APPEND. However, a [BACKSPACE statement](#) or an [OPEN statement](#)
36 containing POSITION = 'APPEND' might fail if the processor needs to read the file to achieve the positioning.

37 7 The following examples illustrate these rules. In the first example, [unit 10](#) is preconnected to a SCRATCH
38 file; the [OPEN statement](#) changes the value of [PAD=](#) to YES.

39 CHARACTER (LEN = 20) CH1

```

1      WRITE (10, '(A)') 'THIS IS RECORD 1'
2      OPEN (UNIT = 10, STATUS = 'OLD', PAD = 'YES')
3      REWIND 10
4      READ (10, '(A20)') CH1    ! CH1 now has the value
5                                ! 'THIS IS RECORD 1'

```

- 6 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.

```

9      CHARACTER (LEN = 25) CH2, CH3
10     OPEN (12, FILE = 'FRED', STATUS = 'OLD', DELIM = 'NONE')
11     CH2 = '"THIS STRING HAS QUOTES."'
12           ! Quotes in string CH2
13     WRITE (12, *) CH2           ! Written with no delimiters
14     OPEN (12, DELIM = 'QUOTE') ! Now quote is the delimiter
15     REWIND 12
16     READ (12, *) CH3           ! CH3 now has the value
17                               ! 'THIS STRING HAS QUOTES.'

```

- 18 9 The next example is invalid because it attempts to change the value of the [STATUS= specifier](#).

```

19     OPEN (10, FILE = 'FRED', STATUS = 'OLD')
20     WRITE (10, *) A, B, C
21     OPEN (10, STATUS = 'SCRATCH') ! Attempts to make FRED a SCRATCH file

```

- 22 10 The previous example could be made valid by closing the [unit](#) first, as in the next example.

```

23     OPEN (10, FILE = 'FRED', STATUS = 'OLD')
24     WRITE (10, *) A, B, C
25     CLOSE (10)
26     OPEN (10, STATUS = 'SCRATCH') ! Opens a different SCRATCH file

```

27 C.8.4 Connection properties (12.5.4)

- 28 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.

- 30 (1) An access method, which is sequential, direct, or stream, is established for the connection
31 ([12.5.6.3](#)).
- 32 (2) A form, which is formatted or unformatted, is established for a connection to a file that ex-
33 ists or is created by the connection. For a connection that results from execution of an [OPEN](#)
34 [statement](#), a default form (which depends on the access method, as described in [12.3.3](#)) is es-
35 tablished if no form is specified. For a preconnected file that exists, a form is established by
36 preconnection. For a preconnected file that does not exist, a form might be established, or the
37 establishment of a form might be delayed until the file is created (for example, by execution of
38 a formatted or unformatted [WRITE statement](#)) ([12.5.6.11](#)).
- 39 (3) A record length might be established. If the access method is direct, the connection establishes
40 a record length that specifies the length of each record of the file. A direct access file can only
41 contain records that are all of equal length.
- 42 (4) A sequential file can contain records of varying lengths. In this case, the record length estab-
43 lished specifies the maximum length of a record in the file ([12.5.6.16](#)).

- 1 2 A processor has wide latitude in adapting these concepts and actions to its own cataloging and job control
2 conventions. Some processors might need job control action to specify the set of files that exist or that will
3 be created by a program. Some processors might not need any job control action prior to execution. This
4 document enables processors to perform dynamic open, close, or file creation operations, but it does not
5 require such capabilities of the processor.
- 6 3 The meaning of “open” in contexts other than Fortran might include such things as mounting a tape, console
7 messages, spooling, label checking, security checking, etc. These actions might occur upon job control
8 action external to Fortran, upon execution of an **OPEN statement**, or upon execution of the first read or
9 write of the file. The **OPEN statement** describes properties of the connection to the file and might or might
10 not cause physical activities to take place.

11 **C.8.5 Asynchronous input/output (12.6.2.5)**

- 12 1 Rather than limit support for asynchronous input/output to what has been traditionally provided by fa-
13 cilities such as BUFFERIN/BUFFEROUT, this document builds upon existing Fortran syntax. This permits
14 alternative approaches for implementing asynchronous input/output, and simplifies the task of adapting
15 existing standard-conforming programs to use asynchronous input/output.
- 16 2 Not all processors actually perform input/output asynchronously, nor will every processor that does be
17 able to handle **data transfer statements** with complicated input/output item lists in an asynchronous man-
18 ner. Such processors can still be standard-conforming.
- 19 3 This document allows for at least two different conceptual models for asynchronous input/output.
- 20 4 Model 1: the processor performs asynchronous input/output when the item list is simple (perhaps one
21 contiguous named array) and the input/output is unformatted. The implementation cost is reduced, and
22 this is the scenario most likely to be beneficial on traditional “big-iron” machines.
- 23 5 Model 2: The processor is free to do any of the following:
 - 24 (1) on output, create a buffer inside the input/output library, completely formatted, and then start
25 an asynchronous write of the buffer, and immediately return to the next statement in the pro-
26 gram. The processor is free to wait for previously issued WRITES, or not, or
 - 27 (2) pass the input/output list addresses to another processor/process, which processes the list
28 items independently of the processor that executes the user’s code. The addresses of the list
29 items will need to be computed before the asynchronous **READ/WRITE** statement completes.
30 There is still an ordering requirement on list item processing to handle things like **READ (...)**
31 **N,(a(i),i=1,N).**
- 32 6 A program can issue a large number of asynchronous input/output requests, without waiting for any of
33 them to complete, and then wait for any or all of them. That does not constitute a requirement for the
34 processor to keep track of each individual request separately.
- 35 7 It is not necessary for all requests to be tracked by the runtime library. If an **ID= specifier** does not appear
36 in on a **READ** or **WRITE** statement, the runtime library can forget about this particular request once it
37 has successfully completed. If an error or end-of-file condition occurs for a request, the processor can
38 report this during any input/output operation to that **unit**. If an **ID= specifier** appears, the processor’s
39 runtime input/output library will need to keep track of any end-of-file or error conditions for that particular
40 input/output request. However, if the input/output request succeeds without any exceptional conditions
41 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 **WAIT**s 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.9 Clause 13 notes

C.9.1 Number of records (13.4, 13.5, 13.8.2)

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.9.2 List-directed input (13.10.3)

1 The following examples illustrate list-directed input. A blank character is represented by b.

2 Example 1:

Program:

```

1      J = 3
2      READ *, I
3      READ *, J

```

4 Sequential input file:

```

5      record 1:  b1b,4bbbbbb
6      record 2:  ,2bbbbbbbbb

```

7 3 Result: I = 1, J = 3.

8 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.

10 5 Example 2:

11 Program:

```

12      CHARACTER A *8, B *1
13      READ *, A, B

```

14 Sequential input file:

```

15      record 1:  'bbbbbbbbb'
16      record 2:  'QXY'b'Z'

```

17 6 Result: A = 'bbbbbbbbb', B = 'Q'

18 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 'bbbbbbbbb'. The end of a record acts as a blank, which in this case is a value separator because it occurs between two constants.

23 C.10 Clause [14](#) notes

24 C.10.1 Main program and block data program unit ([14.1](#), [14.3](#))

- 25 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.
- 27 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.

30 C.10.2 Dependent compilation ([14.2](#))

31 C.10.2.1 Separate translation

- 32 1 This document, like its predecessors, is intended to enable the implementation of conforming processors in which a program can be broken into multiple units, each of which can be separately translated in preparation for execution. Such processors are commonly described as supporting separate compilation. There is

an important difference between the way separate compilation can be implemented under this document and the way it could be implemented under the FORTRAN 77 International Standard. Under the FORTRAN 77 standard, any information required to translate a **program unit** was specified in that **program unit**. Each translation was thus totally independent of all others. Under this document, a **program unit** can use information that was specified in a separate module and thus can be dependent on that module. The implementation of this dependency in a processor might be that the translation of a **program unit** depends on the results of translating one or more modules. Processors implementing the dependency this way are commonly described as supporting dependent compilation.

The dependencies involved here are new only in the sense that the Fortran processor is now aware of them. The same information dependencies existed under the FORTRAN 77 International Standard, but it was the programmer's responsibility to transport the information necessary to resolve them by making redundant specifications of the information in multiple **program units**. The availability of separate but dependent compilation offers several potential advantages over the redundant textual specification of information.

- (1) Specifying information at a single place in the program ensures that different **program units** using that information are translated consistently. Redundant specification leaves the possibility that different information can be erroneously be specified. Even if an **INCLUDE** line is used to ensure that the text of the specifications is identical in all involved **program units**, the presence of other specifications (for example, an **IMPLICIT statement**) could change the interpretation of that text.
- (2) During the revision of a program, it is possible for a processor to assist in determining whether different **program units** have been translated using different (incompatible) versions of a module, although there is no requirement that a processor provide such assistance. Inconsistencies in redundant textual specification of information, on the other hand, tend to be much more difficult to detect.
- (3) Putting information in a module provides a way of packaging it. Without modules, redundant specifications frequently are interleaved with other specifications in a **program unit**, making convenient packaging of such information difficult.
- (4) Because a processor can be implemented such that the specifications in a module are translated once and then repeatedly referenced, there is the potential for greater efficiency than when the processor translates redundant specifications of information in multiple **program units**.

The exact meaning of the requirement that the public portions of a module be available at the time of reference is processor dependent. For example, a processor could consider a module to be available only after it has been compiled and require that if the module has been compiled separately, the result of that compilation be identified to the compiler when compiling **program units** that use it.

C.10.2.2 **USE statement and dependent compilation (14.2.2)**

Another benefit of the **USE statement** is its enhanced facilities for name management. If one needs to use only selected entities in a module, one can do so without having to worry about the names of all the other entities in that module. If one needs to use two different modules that happen to contain entities with the same name, there are several ways to deal with the conflict. If none of the entities with the same name are to be used, they can simply be ignored. If the name happens to refer to the same entity in both modules (for example, if both modules obtained it from a third module), then there is no confusion about what the name denotes and the name can be freely used. If the entities are different and one or both is to be used, the local renaming facility in the **USE statement** makes it possible to give those entities different names in the **program unit** containing the **USE statements**.

- 1 2 A benefit of using the **ONLY** option consistently, as compared to **USE** without it, is that the module from
 2 which each accessed entity is accessed is explicitly specified in each **program unit**. This means that one
 3 need not search other **program units** to find where each one is defined. This reduces maintenance costs.
- 4 3 A typical implementation of dependent but separate compilation might involve storing the result of trans-
 5 lating a module in a file whose name is derived from the name of the module. Note, however, that the name
 6 of a module is limited only by the Fortran rules and not by the names allowed in the file system. Thus the
 7 processor might have to provide a mapping between Fortran names and file system names.
- 8 4 The result of translating a module could reasonably either contain only the information textually specified
 9 in the module (with “pointers” to information originally textually specified in other modules) or contain
 10 all information specified in the module (including copies of information originally specified in other mod-
 11 ules). Although the former approach would appear to save on storage space, the latter approach can greatly
 12 simplify the logic necessary to process a **USE statement** and can avoid the necessity of imposing a limit on
 13 the logical “nesting” of modules via the **USE statement**.
- 14 5 There is an increased potential for undetected errors in a **scoping unit** that uses both implicit typing and
 15 the **USE statement**. For example, in the program fragment

```

16      SUBROUTINE SUB
17          USE MY_MODULE
18          IMPLICIT INTEGER (I-N), REAL (A-H, O-Z)
19          X = F (B)
20          A = G (X) + H (X + 1)
21      END SUBROUTINE SUB
  
```

22 X could be either an implicitly typed real variable or a variable obtained from the module MY_MODULE and
 23 might change from one to the other because of changes in MY_MODULE unrelated to the action performed
 24 by SUB. Logic errors resulting from this kind of situation can be extremely difficult to locate. Thus, the use
 25 of these features together is discouraged.

26 C.10.2.3 Accessibility attributes (8.5.2)

- 27 1 The **PUBLIC** and **PRIVATE** attributes, which can be declared only in modules, divide the entities in a module
 28 into those that are actually relevant to a **scoping unit** referencing the module and those that are not. This
 29 information might be used to improve the performance of a Fortran processor. For example, it might be
 30 possible to discard much of the information about the private entities once a module has been translated,
 31 thus saving on both storage and the time to search it. Similarly, it might be possible to recognize that two
 32 versions of a module differ only in the private entities they contain and avoid retranslating **program units**
 33 that use that module when switching from one version of the module to the other.

34 C.10.3 Examples of the use of modules (14.2.1)

35 C.10.3.1 Global data (14.2.1)

- 36 1 A module could contain only data objects, for example:

```

37      MODULE DATA_MODULE
38          SAVE
39          REAL A (10), B, C (20,20)
40          INTEGER :: I=0
41          INTEGER, PARAMETER :: J=10
42          COMPLEX D (J,J)
43      END MODULE DATA_MODULE
  
```

1 2 Data objects made global in this manner can have any combination of data types.

2 3 Access to some of these can be made by a [USE statement](#) with the [ONLY](#) option, such as:

```
3      USE DATA_MODULE, ONLY: A, B, D
```

4 and access to all of them can be made by the following [USE statement](#):

```
5      USE DATA_MODULE
```

6 4 Access to all of them with some renaming to avoid name conflicts can be made by, for example:

```
7      USE DATA_MODULE, AMODULE => A, DMODULE => D
```

8 C.10.3.2 Derived types ([14.2.1](#))

9 1 A derived type can be defined in a module and accessed in a number of [program units](#). For example,

```
10      MODULE SPARSE
11          TYPE NONZERO
12              REAL A
13              INTEGER I, J
14          END TYPE NONZERO
15      END MODULE SPARSE
```

16 defines a type consisting of a real component and two integer components for holding the numerical value
17 of a nonzero matrix element and its row and column indices.

18 C.10.3.3 Global allocatable arrays ([14.2.1](#))

19 1 Many programs need large global allocatable arrays whose sizes are not known before program execution.
20 A simple form for such a program is:

```
21      PROGRAM GLOBAL_WORK
22          CALL CONFIGURE_ARRAYS      ! Perform the appropriate allocations
23          CALL COMPUTE               ! Use the arrays in computations
24      END PROGRAM GLOBAL_WORK
25      MODULE WORK_ARRAYS            ! An example set of work arrays
26          INTEGER N
27          REAL, ALLOCATABLE :: A (:), B (:, :), C (:, :, :)
28      END MODULE WORK_ARRAYS
29      SUBROUTINE CONFIGURE_ARRAYS    ! Process to set up work arrays
30          USE WORK_ARRAYS
31          READ (*, *) N
32          ALLOCATE (A (N), B (N, N), C (N, N, 2 * N))
33      END SUBROUTINE CONFIGURE_ARRAYS
34      SUBROUTINE COMPUTE
35          USE WORK_ARRAYS
36          ... Computations involving arrays A, B, and C.
37      END SUBROUTINE COMPUTE
```

38 2 Typically, many subprograms need access to the work arrays, and all such subprograms would contain the
39 statement

```
40      USE WORK_ARRAYS
```

1 C.10.3.4 Procedure libraries (14.2.2)

1 Interface bodies for external procedures in a library can be gathered into a module. An interface body specifies an explicit interface (15.4.2.2).

2 An example is the following library module:

```

5      MODULE LIBRARY_LLS
6          INTERFACE
7              SUBROUTINE LLS (X, A, F, FLAG)
8                  REAL X (:, :)
9                  ! The SIZE in the next statement is an intrinsic function
10                 REAL, DIMENSION (SIZE (X, 2)) :: A, F
11                 INTEGER FLAG
12             END SUBROUTINE LLS
13         ...
14     END INTERFACE
15     ...
16 END MODULE LIBRARY_LLS

```

3 This module provides an explicit interface that is necessary for the subroutine LLS to be invoked. for example:

```

19      USE LIBRARY_LLS
20      ...
21      CALL LLS (X = ABC, A = D, F = XX, FLAG = IFLAG)
22      ...

```

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.

26 C.10.3.5 Operator extensions (14.2.2)

1 In order to extend an intrinsic operator symbol to have an additional meaning, an interface block specifying that operator symbol in the OPERATOR option of the INTERFACE statement could be placed in a module.

2 For example, // can be extended to perform concatenation of two derived-type objects serving as varying length character strings and + can be extended to specify matrix addition for type MATRIX or interval arithmetic addition for type INTERVAL.

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.

34 C.10.3.6 Data abstraction (14.2.2)

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

```



```

1      DO J = 1, B % CARD
2          IF (.NOT. (B % ELEMENT (J) .IN. A)) THEN
3              IF (UNION % CARD < MAX_SET_CARD) THEN
4                  UNION % CARD = UNION % CARD + 1
5                  UNION % ELEMENT (UNION % CARD) = B % ELEMENT (J)
6              ELSE
7                  ! Maximum set size exceeded . . .
8              END IF
9          END IF
10     END DO
11 END FUNCTION UNION
12
13 FUNCTION DIFFERENCE (A, B)                ! Difference of sets A and B
14     TYPE (SET) DIFFERENCE
15     TYPE (SET), INTENT(IN) :: A, B
16     INTEGER J, X
17     DIFFERENCE % CARD = 0                ! The empty set
18     DO J = 1, A % CARD
19         X = A % ELEMENT (J)
20         IF (.NOT. (X .IN. B)) DIFFERENCE = DIFFERENCE + SET (1, X)
21     END DO
22 END FUNCTION DIFFERENCE
23
24 FUNCTION INTERSECTION (A, B)              ! Intersection of sets A and B
25     TYPE (SET) INTERSECTION
26     TYPE (SET), INTENT(IN) :: A, B
27     INTERSECTION = A - (A - B)
28 END FUNCTION INTERSECTION
29
30 LOGICAL FUNCTION SUBSET (A, B)            ! Determines if set A is
31     TYPE (SET), INTENT(IN) :: A, B        ! a subset of set B
32     INTEGER I
33     SUBSET = A % CARD <= B % CARD
34     IF (.NOT. SUBSET) RETURN              ! For efficiency
35     DO I = 1, A % CARD
36         SUBSET = SUBSET .AND. (A % ELEMENT (I) .IN. B)
37     END DO
38 END FUNCTION SUBSET
39
40 TYPE (SET) FUNCTION SETF (V)              ! Transfer function between a vector
41     INTEGER V (:)                        ! of elements and a set of elements
42     INTEGER J                            ! removing duplicate elements
43     SETF % CARD = 0
44     DO J = 1, SIZE (V)
45         IF (.NOT. (V (J) .IN. SETF)) THEN
46             IF (SETF % CARD < MAX_SET_CARD) THEN
47                 SETF % CARD = SETF % CARD + 1
48                 SETF % ELEMENT (SETF % CARD) = V (J)
49             ELSE
50                 ! Maximum set size exceeded . . .
51             END IF
52         END IF
53     END DO
54 END FUNCTION SETF
55
56 FUNCTION VECTOR (A)                      ! Transfer the values of set A
57     TYPE (SET), INTENT (IN) :: A        ! into a vector in ascending order
58     INTEGER, POINTER :: VECTOR (:)
59     INTEGER I, J, K
60     ALLOCATE (VECTOR (A % CARD))
61     VECTOR = A % ELEMENT (1 : A % CARD)

```

```

1      DO I = 1, A % CARD - 1          ! Use a better sort if
2      DO J = I + 1, A % CARD          ! A % CARD is large
3      IF (VECTOR (I) > VECTOR (J)) THEN
4      K = VECTOR (J); VECTOR (J) = VECTOR (I); VECTOR (I) = K
5      END IF
6      END DO
7      END DO
8      END FUNCTION VECTOR
9      END MODULE INTEGER_SETS

```

10 4 Examples of using INTEGER_SETS (A, B, and C are variables of type SET; X is an integer variable):

```

11      ! Check to see if A has more than 10 elements
12      IF (CARDINALITY (A) > 10) ...
13
14      ! Check for X an element of A but not of B
15      IF (X .IN. (A - B)) ...
16
17      ! C is the union of A and the result of B intersected
18      ! with the integers 1 to 100
19      C = A + B * SETF ([ (I, I = 1, 100) ])
20
21      ! Does A have any even numbers in the range 1:100?
22      IF (CARDINALITY (A * SETF ([ (I, I = 2, 100, 2) ])) > 0) ...
23
24      PRINT *, VECTOR (B) ! Print out the elements of set B, in ascending order

```

25 C.10.3.7 Public entities renamed (14.2.2)

- 26 1 At times it might be necessary to rename entities that are accessed with [USE statements](#).
- 27 2 The following example illustrates renaming features of the [USE statement](#).

```

28      MODULE J; REAL JX, JY, JZ; END MODULE J
29      MODULE K
30      USE J, ONLY : KX => JX, KY => JY
31      ! KX and KY are local names to module K
32      REAL KZ          ! KZ is local name to module K
33      REAL JZ          ! JZ is local name to module K
34      END MODULE K
35      PROGRAM RENAME
36      USE J; USE K
37      ! Module J's entity JX is accessible under names JX and KX
38      ! Module J's entity JY is accessible under names JY and KY
39      ! Module K's entity KZ is accessible under name KZ
40      ! Module J's entity JZ and K's entity JZ are different entities
41      ! and cannot be referenced
42      ...
43      END PROGRAM RENAME

```

44 C.10.4 Modules with submodules (14.2.3)

- 45 1 Each submodule specifies that it is the child of exactly one parent module or submodule. Therefore, a
46 module and all of its descendant submodules stand in a tree-like relationship one to another.
- 47 2 A separate module procedure that is declared in a module to have public accessibility can be accessed by
48 use association even if it is defined in a submodule. No other entity in a submodule can be accessed by
49 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 re-translation 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

```

```

1           ! color_points is in the color_points_a submodule.
2       type(palette), intent(in) :: p
3       ...
4       end subroutine test_palette
5   end module palette_stuff
6
7   submodule ( color_points:color_points_a ) color_points_b ! Subsidiary**2 submodule
8
9   contains
10      ! Invisible body for interface declared in the ancestor module
11      module subroutine color_point_draw ( p )
12          use palette_stuff, only: palette
13          type(color_point), intent(in) :: p
14          type(palette) :: MyPalette
15          ...; call inquire_palette ( p, MyPalette ); ...
16      end subroutine color_point_draw
17
18      ! Invisible body for interface declared in the parent submodule
19      module procedure inquire_palette
20          ... Implementation of inquire_palette.
21      end procedure inquire_palette
22
23      subroutine private_stuff ! not accessible from color_points_a
24          ...
25      end subroutine private_stuff
26
27 end submodule color_points_b

```

- 28 9 There is a use palette_stuff in color_points_a, and a use color_points in palette_stuff. The use
 29 palette_stuff would cause a circular reference if it appeared in color_points. In this case, it does not
 30 cause a circular dependence because it is in a submodule. Submodules cannot be referenced by use asso-
 31 ciation, and therefore what would be a circular appearance of use palette_stuff is not accessed.

```

32 program main
33     use color_points
34     ! "instance_count" and "inquire_palette" are not accessible here
35     ! because they are not declared in the "color_points" module.
36     ! "color_points_a" and "color_points_b" cannot be referenced by
37     ! use association.
38     interface draw ! just to demonstrate it's possible
39         module procedure color_point_draw
40     end interface
41     type(color_point) :: C_1, C_2
42     real :: RC
43     ...
44     call color_point_new (c_1) ! body in color_points_a,
45                               ! interface in color_points
46     ...
47     call draw (c_1) ! body in color_points_b, specific interface
48                   ! in color_points, generic interface here.
49     ...
50     rc = color_point_dist (c_1, c_2) ! body in color_points_a,
51                                     ! interface in color_points
52     ...
53     call color_point_del (c_1) ! body in color_points_a,
54                               ! interface in color_points
55     ...
56 end program main

```

- 57 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.

Consider a `Plasma` module from a Tokamak 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		

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.11 Clause 15 notes

C.11.1 Portability problems with external procedures (15.4.3.5)

There is a potential portability problem in a [scoping unit](#) that references an [external procedure](#) without explicitly declaring it to have the [EXTERNAL attribute](#) (8.5.9). On a different processor, the name of that procedure might be the name of a nonstandard intrinsic procedure and in such a case the processor would interpret those procedure references as references to that intrinsic procedure. (On that processor, the program would also be viewed as not conforming to this document because of the references to the nonstandard intrinsic procedure.) Declaration of the [EXTERNAL attribute](#) causes the references to be to the [external procedure](#) regardless of the availability of an intrinsic procedure with the same name. Note that declaration of the type of a procedure is not enough to make it [external](#), even if the type is inconsistent with the type of the result of an intrinsic procedure of the same name.

C.11.2 Procedures defined by means other than Fortran (15.6.3)

A processor is not required to provide any means other than Fortran for defining [external procedures](#). Among the means that might be supported are the machine assembly language, other high level languages, the Fortran language extended with nonstandard features, and the Fortran language as supported by another Fortran processor (for example, a previously existing FORTRAN 77 processor). The means other than Fortran for defining [external procedures](#), including any restrictions on the structure or organization of those procedures, are not specified by this document.

A Fortran processor might limit its support of procedures defined by means other than Fortran such that these procedures can affect entities in the Fortran environment only on the same basis as procedures written in Fortran. For example, it might not support the value of a local variable from being changed by a procedure reference unless that variable were one of the arguments to the procedure.

C.11.3 Abstract interfaces and procedure pointer components (15.4, 7.5)

- 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
    TYPE(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
    TYPE(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
    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(callback%data)
      DEALLOCATE(callback%data)
    ELSE
      CALL callback%proc
    END IF
    DEALLOCATE(callback)
  END SUBROUTINE

```



```

1      END SUBROUTINE
2      !
3      ! Call the procedures in reverse order of registration
4      !
5      SUBROUTINE invoke_callback_reverse(list)
6          TYPE(callback_list),INTENT(INOUT) :: list
7          TYPE(callback_record),POINTER :: next,current
8          current => list%first
9          NULLIFY(list%first)
10         DO WHILE (ASSOCIATED(current))
11             next => current%next
12             CALL invoke(current)
13             current => next
14         END DO
15     END SUBROUTINE
16     !
17     ! Internal: Forward mode invocation
18     !
19     SUBROUTINE forward_invoke(callback)
20         TYPE(callback_record),POINTER :: callback
21         IF (ASSOCIATED(callback%next)) CALL forward_invoke(callback%next)
22         CALL invoke(callback)
23     END SUBROUTINE
24     !
25     ! Call the procedures in forward order of registration
26     !
27     SUBROUTINE invoke_callback_forward(list)
28         TYPE(callback_list),INTENT(INOUT) :: list
29         IF (ASSOCIATED(list%first)) CALL forward_invoke(list%first)
30     END SUBROUTINE
31 END

```

C.11.4 Pointers and targets as arguments (15.5.2.5, 15.5.2.7, 15.5.2.8)

- 1 If a dummy argument is declared to be a **pointer**, the corresponding **actual argument** could be a **pointer** or could be a nonpointer variable or procedure. Consider the two cases separately.

Case (i): The **actual argument** is a pointer. When procedure execution commences the **pointer association** status of the **dummy argument** becomes the same as that of the **actual argument**. If the **pointer association** status of the **dummy argument** is changed, the **pointer association** status of the **actual argument** changes in the same way.

Case (ii): The **actual argument** is not a pointer. This only occurs when the **actual argument** has the **TARGET attribute** or is a procedure, and the **dummy argument** has the **INTENT (IN) attribute**. The **dummy argument** becomes pointer associated with the **actual argument**.

- 2 When execution of a procedure completes, any **data 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) ! On return PBEST is associated with the 'best' element of B.
...
CONTAINS
SUBROUTINE BEST (P, A)

```

```

1      REAL, POINTER, INTENT (OUT)  :: P
2      REAL, TARGET, INTENT (IN)    :: A (:)
3      ... Find the "best" element A(I).
4      P => A (I)
5      END SUBROUTINE BEST
6      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:

```

11     INTEGER LARGE(100,100)
12     CALL SUB (LARGE)
13     ...
14     CALL SUB ( )
15     CONTAINS
16     SUBROUTINE SUB(ARG)
17         INTEGER, TARGET, OPTIONAL :: ARG(100,100)
18         INTEGER, POINTER, DIMENSION(:, :) :: PARG
19         IF (PRESENT(ARG)) THEN
20             PARG => ARG
21         ELSE
22             ALLOCATE (PARG(100,100))
23             PARG = 0
24         ENDIF
25         ... Code with lots of references to PARG.
26         IF (.NOT. PRESENT(ARG)) DEALLOCATE (PARG)
27     END SUBROUTINE SUB
28     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.11.5 Polymorphic Argument Association (15.5.2.10)

- 1 The following example illustrates the polymorphic **argument association** rules using the derived types defined in 7.5.7.2, NOTE 4.

```

38     TYPE(POINT) :: T2
39     TYPE(COLOR_POINT) :: T3
40     CLASS(POINT) :: P2
41     CLASS(COLOR_POINT) :: P3
42     ! Dummy argument is polymorphic and actual argument is of fixed type
43     SUBROUTINE SUB2 ( X2 ); CLASS(POINT) :: X2; ...
44     SUBROUTINE SUB3 ( X3 ); CLASS(COLOR_POINT) :: X3; ...
45
46     CALL SUB2 ( T2 ) ! Valid -- The declared type of T2 is the same as the
47                     !           declared type of X2.
48     CALL SUB2 ( T3 ) ! Valid -- The declared type of T3 is extended from
49                     !           the declared type of X2.
50     CALL SUB3 ( T2 ) ! Invalid -- The declared type of T2 is neither the
51                     !           same as nor extended from the declared type

```

```

1           !           type of X3.
2   CALL SUB3 ( T3 ) ! Valid -- The declared type of T3 is the same as the
3           !           declared type of X3.
4   ! Actual argument is polymorphic and dummy argument is of fixed type
5   SUBROUTINE TUB2 ( D2 ); TYPE(POINT) :: D2; ...
6   SUBROUTINE TUB3 ( D3 ); TYPE(COLOR_POINT) :: D3; ...
7
8   CALL TUB2 ( P2 ) ! Valid -- The declared type of P2 is the same as the
9           !           declared type of D2.
10  CALL TUB2 ( P3 ) ! Invalid -- The declared type of P3 differs from the
11           !           declared type of D2.
12  CALL TUB2 ( P3%POINT ) ! Valid alternative to the above
13  CALL TUB3 ( P2 ) ! Invalid -- The declared type of P2 differs from the
14           !           declared type of D3.
15  SELECT TYPE ( P2 ) ! Valid conditional alternative to the above
16  CLASS IS ( COLOR_POINT ) ! Works if the dynamic type of P2 is the same
17      CALL TUB3 ( P2 )      ! as the declared type of D3, or a type
18                           ! extended therefrom.
19  CLASS DEFAULT
20                           ! Cannot work if not.
21  END SELECT
22  CALL TUB3 ( P3 ) ! Valid -- The declared type of P3 is the same as the
23           !           declared type of D3.
24  ! Both the actual and dummy arguments are of polymorphic type.
25  CALL SUB2 ( P2 ) ! Valid -- The declared type of P2 is the same as the
26           !           declared type of X2.
27  CALL SUB2 ( P3 ) ! Valid -- The declared type of P3 is extended from
28           !           the declared type of X2.
29  CALL SUB3 ( P2 ) ! Invalid -- The declared type of P2 is neither the
30           !           same as nor extended from the declared
31           !           type of X3.
32  SELECT TYPE ( P2 ) ! Valid conditional alternative to the above
33  CLASS IS ( COLOR_POINT ) ! Works if the dynamic type of P2 is the
34      CALL SUB3 ( P2 )      ! same as the declared type of X3, or a
35                           ! type extended therefrom.
36  CLASS DEFAULT
37                           ! Cannot work if not.
38  END SELECT
39  CALL SUB3 ( P3 ) ! Valid -- The declared type of P3 is the same as the
40           !           declared type of X3.

```

C.11.6 Rules ensuring unambiguous generics (15.4.3.4.5)

- 1 The rules in 15.4.3.4.5 are intended to ensure
 - that it is possible to reference each specific procedure or binding in the generic collection,
 - that for any valid generic procedure reference, the determination of the specific procedure referenced is unambiguous, and
 - that the determination of the specific procedure or binding referenced can be made before execution of the program begins (during compilation).
- 2 Interfaces of specific procedures or bindings are distinguished by fixed properties of their arguments, specifically type, kind type parameters, [rank](#), and whether the dummy argument has the [POINTER](#) or [ALLOCATABLE](#) attribute. A valid reference to one procedure in a generic collection will differ from another because it has an argument that the other cannot accept, because it is missing an argument that the other requires, or because one of these fixed properties is different.

- 1 3 Although the declared type of a data entity is a fixed property, polymorphic variables allow for a limited
2 degree of type mismatch between dummy arguments and **actual arguments**, so the requirement for distin-
3 guishing two dummy arguments is type incompatibility, not merely different types. (This is illustrated in
4 the `BAD6` example later in this subclause.)
- 5 4 That same limited type mismatch means that two dummy arguments that are not type incompatible can be
6 distinguished on the basis of the values of the kind type parameters they have in common; if one of them
7 has a kind type parameter that the other does not, that is irrelevant in distinguishing them.
- 8 5 **Rank** is a fixed property, but some forms of array dummy arguments allow **rank** mismatches when a proced-
9 ure is referenced by its specific name. In order to allow **rank** to always be usable in distinguishing generics,
10 such **rank** mismatches are disallowed for those arguments when the procedure is referenced as part of a
11 generic. Additionally, the fact that elemental procedures can accept array arguments is not taken into ac-
12 count when applying these rules, so apparent ambiguity between elemental and nonelemental procedures
13 is possible; in such cases, the reference is interpreted as being to the nonelemental procedure.
- 14 6 For procedures referenced as operators or defined-assignment, syntactically distinguished arguments are
15 mapped to specific positions in the argument list, so the rule for distinguishing such procedures is that it
16 be possible to distinguish the arguments at one of the argument positions.
- 17 7 For **defined input/output** procedures, only the `dtv` argument corresponds to something explicitly written
18 in the program, so it is the `dtv` that is required to be distinguished. Because `dtv` arguments are required to
19 be scalar, they cannot differ in **rank**. Thus this rule effectively involves only type and kind type parameters.
- 20 8 For generic procedure names, the rules are more complicated because optional arguments can be omitted
21 and because arguments can be specified either positionally or by name.
- 22 9 In the special case of type-bound procedures with **passed-object dummy arguments**, the passed-object
23 argument is syntactically distinguished in the reference, so rule (3) in 15.4.3.4.5 can be applied. The type of
24 passed-object arguments is constrained in ways that prevent passed-object arguments in the same **scoping**
25 **unit** from being type incompatible. Thus this rule effectively involves only kind type parameters and **rank**.
- 26 10 The primary means of distinguishing named generics is rule (4). The most common application of that rule
27 is a single argument satisfying both (4a) and (4b):

```

28     INTERFACE GOOD1
29         FUNCTION F1A(X)
30             REAL :: F1A,X
31         END FUNCTION F1A
32         FUNCTION F1B(X)
33             INTEGER :: F1B,X
34         END FUNCTION F1B
35     END INTERFACE GOOD1

```

- 36 11 Whether one writes `GOOD1(1.0)` or `GOOD1(X=1.0)`, the reference is to `F1A` because `F1B` would require an
37 integer argument whereas these references provide the real constant 1.0.
- 38 12 This example and those that follow are expressed using interface bodies, with type as the distinguishing
39 property. This was done to make it easier to write and describe the examples. The principles being illus-
40 trated are equally applicable when the procedures get their explicit interfaces in some other way or when
41 kind type parameters or **rank** are the distinguishing property.
- 42 13 Another common variant is the argument that satisfies (4a) and (4b) by being required in one specific and
43 completely missing in the other:

```

1      INTERFACE GOOD2
2          FUNCTION F2A(X)
3              REAL :: F2A,X
4          END FUNCTION F2A
5          FUNCTION F2B(X,Y)
6              COMPLEX :: F2B
7              REAL :: X,Y
8          END FUNCTION F2B
9      END INTERFACE GOOD2

```

10 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`,
11 because `F2A` has no argument in the second position or with the name `Y`. This approach is used as an altern-
12 ative to optional arguments when one wants a function to have different result type, kind type parameters,
13 or [rank](#), depending on whether the argument is present. In many of the intrinsic functions, the `DIM` argu-
14 ment works this way.

15 15 It is possible to construct cases where different arguments are used to distinguish positionally and by name:

```

16      INTERFACE GOOD3
17          SUBROUTINE S3A(W,X,Y,Z)
18              REAL :: W,Y
19              INTEGER :: X,Z
20          END SUBROUTINE S3A
21          SUBROUTINE S3B(X,W,Z,Y)
22              REAL :: W,Z
23              INTEGER :: X,Y
24          END SUBROUTINE S3B
25      END INTERFACE GOOD3

```

26 16 If one writes `GOOD3(1.0,2,3.0,4)` to reference `S3A`, then the third and fourth arguments are consistent with
27 a reference to `S3B`, but the first and second are not. If one switches to writing the first two arguments as
28 keyword arguments in order for them to be consistent with a reference to `S3B`, the latter two arguments will
29 also need to be written as keyword arguments, `GOOD3(X=2,W=1.0,Z=4,Y=3.0)`, and the named arguments
30 `Y` and `Z` are distinguished.

31 17 The ordering requirement in rule (4) is critical:

```

32      INTERFACE BAD4 ! this interface is invalid !
33          SUBROUTINE S4A(W,X,Y,Z)
34              REAL :: W,Y
35              INTEGER :: X,Z
36          END SUBROUTINE S4A
37          SUBROUTINE S4B(X,W,Z,Y)
38              REAL :: X,Y
39              INTEGER :: W,Z
40          END SUBROUTINE S4B
41      END INTERFACE BAD4

```

42 18 In this example, the positionally distinguished arguments are `Y` and `Z`, and it is `W` and `X` that are distinguished
43 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
44 `S4A` and `S4B`.

45 19 Rule (1) can be used to distinguish some cases that are not covered by rule (4):

```

46      INTERFACE GOOD5
47          SUBROUTINE S5A(X)
48              REAL :: X

```

```

1      END SUBROUTINE S5A
2      SUBROUTINE S5B(Y,X)
3          REAL :: Y,X
4      END SUBROUTINE S5B
5  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 (1) 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

```

13      INTERFACE BAD6 ! this interface is invalid !
14          SUBROUTINE S6A(X,Y)
15              CLASS(PEAR) :: X,Y
16          END SUBROUTINE S6A
17          SUBROUTINE S6B(X,Y)
18              CLASS(FRUIT) :: X
19              CLASS(BOSC) :: Y
20          END SUBROUTINE S6B
21      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:

```

25      INTERFACE GOOD7
26          SUBROUTINE S7A(X,Y,Z)
27              CLASS(PEAR) :: X,Y,Z
28          END SUBROUTINE S7A
29          SUBROUTINE S7B(X,Z,W)
30              CLASS(FRUIT) :: X
31              CLASS(BOSC) :: Z
32              CLASS(APPLE),OPTIONAL :: W
33          END SUBROUTINE S7B
34      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:

```

1      INTERFACE BAD8 ! this interface is invalid !
2          ! despite the fact that it is unambiguous !
3      SUBROUTINE S8A(X,Y,Z)
4          REAL,OPTIONAL :: X
5          INTEGER :: Y
6          REAL :: Z
7      END SUBROUTINE S8A
8      SUBROUTINE S8B(X,Z,Y)
9          INTEGER,OPTIONAL :: X
10         INTEGER :: Z
11         REAL :: Y
12     END SUBROUTINE S8B
13 END INTERFACE BAD8

```

26 This interface fails rule (4) because there are no required arguments that can be distinguished from the positionally corresponding argument, but in order for the mismatch of the optional arguments not to be relevant, the later arguments need to be specified as keyword arguments, so distinguishing by name does the trick. This interface is nevertheless invalid so a standard-conforming Fortran processor is not required to do such reasoning. The rules to cover all cases are too complicated to be useful.

27 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.

23 C.12 Clause 16 notes

24 C.12.1 Atomic memory consistency

25 C.12.1.1 Relaxed memory model

26 1 Parallel programs sometimes have apparently impossible behavior because data transfers and other messages can be delayed, reordered and even repeated, by hardware, communication software, and caching and other forms of optimization. Requiring processors to deliver globally consistent behavior is incompatible with performance on many systems. This document specifies that all ordered actions will be consistent (5.3.5 and 11.7), but all consistency between unordered segments is deliberately left processor dependent. Depending on the hardware, this can be observed even when only two images and one mechanism are involved.

33 C.12.1.2 Examples with atomic operations

34 1 When variables are being referenced (atomically) from segments that are unordered with respect to the segment that is atomically defining or redefining the variables, the results are processor dependent. This supports use of so-called “relaxed memory model” architectures, which can enable more efficient execution on some hardware implementations.

38 2 The following examples assume these declarations:

```

39     MODULE EXAMPLE
40     USE, INTRINSIC :: ISO_FORTRAN_ENV
41     INTEGER(ATOMIC_INT_KIND) :: X [*] = 0, Y [*] = 0, TMP

```


1 3 **Example 1**

2 With $X[j]$ and $Y[j]$ still in their initial state (both zero), image j executes the following sequence of state-

3 ments:

4 CALL `ATOMIC_DEFINE` (X , 1)
 5 CALL `ATOMIC_DEFINE` (Y , 1)

6 and a different image, k , executes the following sequence of statements:

7 DO
 8 CALL `ATOMIC_REF` (TMP , $Y[j]$)
 9 IF ($TMP==1$) EXIT
 10 END DO
 11 CALL `ATOMIC_REF` (TMP , $X[j]$)
 12 PRINT *, TMP

13 4 The final value of TMP on image k could be either 0 or 1. That is, even though image j thinks that it defined
 14 $X[j]$ before it defined $Y[j]$, this ordering is not guaranteed to be observed on image k . There are many
 15 aspects of hardware and software implementation that can cause this effect, but conceptually this example
 16 can be thought of as the change in the value of Y propagating faster through the inter-image connections
 17 than the change in the value of X .

18 5 Even if image j executed the sequence

19 CALL `ATOMIC_DEFINE` (X , 1)
 20 SYNC MEMORY
 21 CALL `ATOMIC_DEFINE` (Y , 1)

22 the same effect could be seen. That is because even though X and Y are defined in ordered segments, the
 23 references from image k are both from a segment that is unordered with respect to image j .

24 6 Only if the reference on image k to $Y[j]$ is in a segment that is ordered after the segment on image j that
 25 defined Y , will TMP be guaranteed to have the value 1.

26 7 **Example 2:**

27 With the initial state of X and Y on image j (i.e. $X[j]$ and $Y[j]$) still being zero, execution of

28 CALL `ATOMIC_REF` (TMP , $X[j]$)
 29 CALL `ATOMIC_DEFINE` ($Y[j]$, 1)
 30 PRINT *, TMP

31 on image k_1 , and execution of

32 CALL `ATOMIC_REF` (TMP , $Y[j]$)
 33 CALL `ATOMIC_DEFINE` ($X[j]$, 1)
 34 PRINT *, TMP

35 on image k_2 , in unordered segments, might print the value 1 both times.

36 8 This can happen by such mechanisms as “load buffering”; one might imagine that what is happening is
 37 that the definitions (`ATOMIC_DEFINE`) are overtaking the references (`ATOMIC_REF`). On some processors
 38 it is possible that insertion of `SYNC MEMORY` statements between the calls to `ATOMIC_REF` and `ATOMIC-`
 39 `DEFINE` might be sufficient to make the output print the value 1 at most one time (or even exactly one time),
 40 but this is still processor dependent unless the `SYNC MEMORY statement` executions cause the relevant
 41 segments on images k_1 and k_2 to be ordered.

1 9 **Example 3:**

2 Because there are no segment boundaries implied by collective subroutines, with the initial state as before,
3 execution of

```
4         IF (THIS_IMAGE ()==1) THEN
5             CALL ATOMIC_DEFINE (X [3], 23)
6             Y = 42
7         END IF
8         CALL CO_BROADCAST (Y, 1)
9         IF (THIS_IMAGE ()==2) THEN
10            CALL ATOMIC_REF (TMP, X [3])
11            PRINT *, Y, TMP
12        END IF
```

13 could print the values 42 and 0.

14 10 **Example 4:**

15 Assuming the declarations

```
16         INTEGER (ATOMIC_INT_KIND) :: X [*] = 0, Z = 0
```

17 the statements

```
18         CALL ATOMIC_ADD (X [1], 1)           ! (A)
19         IF (THIS_IMAGE () == 2) THEN
20             wait: DO
21                 CALL ATOMIC_REF (Z, X [1])    ! (B)
22                 IF (Z == NUM_IMAGES ()) EXIT wait
23             END DO wait                       ! (C)
24         END IF
```

25 will execute the “wait” loop on image 2 until all images have completed statement (A). The updates of X
26 [1] are performed by each image in the same manner, but in an arbitrary order. Because the result from
27 the complete set of updates will eventually become visible by execution of statement (B) for some loop
28 iteration on image 2, the termination condition is guaranteed to be eventually fulfilled, provided that no
29 image failure occurs, every image executes the above code, and no other code is executed in an unordered
30 segment that performs an update to X [1]. Furthermore, if two **SYNC MEMORY** statements are inserted
31 in the above code before statement (A) and after statement (C), respectively, the segment started by the
32 second **SYNC MEMORY** on image 2 is ordered after the segments on all images that end with the first **SYNC**
33 **MEMORY**.

34 **C.12.2 EVENT_QUERY example**

35 1 The following example illustrates the use of events via a program in which image one acts as the controlling
36 image, distributing work items to the other images. Only one work item at a time can be active on a worker
37 image, and each deals with the result (e.g. via input/output) without directly feeding data back to the
38 controlling image.

39 2 Because the work items are not expected to be balanced, the controlling image keeps cycling through the
40 other images to find one that is waiting for work.

41 3 An event is posted by each worker to indicate that it has completed its work item. Since the corresponding
42 variables are needed only on the controlling image, we place them in an allocatable array component of a
43 coarray. An event on each worker is needed for the controlling image to post the fact that it has made a
44 work item available for it.

45 **Example code:**

```

1      PROGRAM work_share
2          USE, INTRINSIC :: ISO_FORTRAN_ENV, ONLY: EVENT_TYPE
3          USE :: mod_work, ONLY:
4              work,
5              create_work_item, & ! Function that creates work item
6              process_item,    & ! Function that processes an item
7              work_done        ! Logical function that returns true
8                              ! if all work has been done.
9
10         TYPE :: worker_type
11             TYPE (EVENT_TYPE), ALLOCATABLE :: free (:)
12         END TYPE
13         TYPE (EVENT_TYPE) :: submit [*] ! Post when work ready for a worker
14         TYPE (worker_type) :: worker [*] ! Post when worker is free
15         TYPE (work) :: work_item [*] ! Holds the data for a work item
16         INTEGER :: count, i, nbusy [*]
17
18         IF (THIS_IMAGE ()==1) THEN
19             ! Get started
20             ALLOCATE (worker%free (2:NUM_IMAGES ()))
21             nbusy = 0 ! This holds the number of workers working
22             DO i = 2, NUM_IMAGES () ! Start the workers working
23                 IF (work_done ()) EXIT
24                 nbusy = nbusy + 1
25                 work_item [i] = create_work_item ()
26                 EVENT POST (submit [i])
27             END DO
28             ! Main work distribution loop
29         main: DO
30             image: DO i = 2, NUM_IMAGES ()
31                 CALL EVENT_QUERY (worker%free (i), count)
32                 IF (count==0) CYCLE image ! Worker is not free
33                 EVENT WAIT (worker%free (i))
34                 nbusy = nbusy - 1
35                 IF (work_done ()) CYCLE
36                 nbusy = nbusy + 1
37                 work_item [i] = create_work_item ()
38                 EVENT POST (submit [i])
39             END DO image
40             IF (nbusy==0) THEN
41                 ! All done. Exit on all images.
42                 DO i = 2, NUM_IMAGES ()
43                     EVENT POST (submit [i])
44                 END DO
45                 EXIT main
46             END IF
47         END DO main
48     ELSE
49         ! Work processing loop
50     worker: DO
51         EVENT WAIT (submit)
52         IF (nbusy[1] == 0) EXIT
53         CALL process_item (work_item)
54         EVENT POST (worker [1]%free (THIS_IMAGE ()))
55     END DO worker
56 END IF
57 END PROGRAM work_share

```

C.12.3 Collective subroutine examples

- 1 The following example computes a dot product of two scalar coarrays using `CO_SUM` to store the result in a noncoarray scalar variable.

```
SUBROUTINE codot (x, y, x_dot_y)
  REAL :: x [*], y [*], x_dot_y
  x_dot_y = x*y
  CALL CO_SUM (x_dot_y)
END SUBROUTINE codot
```

- 2 The function below demonstrates passing a noncoarray dummy argument to `CO_MAX`. The function uses `CO_MAX` to find the maximum value of the dummy argument across all images. Then the function flags all images that hold values matching the maximum. The function then returns the maximum image index for an image that holds the maximum value.

```
FUNCTION find_max (j) RESULT (j_max_location)
  INTEGER, INTENT (IN) :: j
  INTEGER j_max, j_max_location
  j_max = j
  CALL CO_MAX (j_max)
  ! Flag images that hold the maximum j.
  IF (j==j_max) THEN
    j_max_location = THIS_IMAGE ()
  ELSE
    j_max_location = 0
  END IF
  ! Return highest image index associated with a maximal j.
  CALL CO_MAX(j_max_location)
END FUNCTION find_max
```

C.13 Clause 18 notes

C.13.1 Runtime environments (18.1)

- 1 This document allows programs to contain procedures defined by means other than Fortran. That raises the issues of initialization of and interaction between the runtime environments involved.
- 2 Implementations are free to solve these issues as they see fit, provided that
- heap allocation/deallocation (e.g., (DE)ALLOCATE in a Fortran subprogram and malloc/free in a C 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.13.2 Example of Fortran calling C (18.3)

C Function Prototype:

```
int C_Library_Function(void* sendbuf, int sendcount, int *recvcounts);
```

Fortran Module:

```

1      MODULE CLIBFUN_INTERFACE
2      INTERFACE
3      INTEGER (C_INT) FUNCTION C_LIBRARY_FUNCTION (SENDBUF, SENDCOUNT, RECVCOUNTS) &
4          BIND(C, NAME='C_Library_Function')
5      USE, INTRINSIC :: ISO_C_BINDING
6      IMPLICIT NONE
7      TYPE (C_PTR), VALUE :: SENDBUF
8      INTEGER (C_INT), VALUE :: SENDCOUNT
9      INTEGER (C_INT) :: RECVCOUNTS(*)
10     END FUNCTION C_LIBRARY_FUNCTION
11 END INTERFACE
12
13 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 shows how `C_Library_Function` might be referenced in a Fortran `program unit`:

```

24     USE, INTRINSIC :: ISO_C_BINDING, ONLY: C_INT, C_FLOAT, C_LOC
25     USE CLIBFUN_INTERFACE
26     ...
27     REAL (C_FLOAT), TARGET :: SEND(100)
28     INTEGER (C_INT) :: SENDCOUNT, RET
29     INTEGER (C_INT), ALLOCATABLE :: RECVCOUNTS(:)
30     ...
31     ALLOCATE( RECVCOUNTS(100) )
32     ...
33     RET = C_LIBRARY_FUNCTION(C_LOC(SEND), SENDCOUNT, RECVCOUNTS)
34     ...

```

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.13.3 Example of C calling Fortran (18.3)

Fortran Code:

```

SUBROUTINE SIMULATION(ALPHA, BETA, GAMMA, DELTA, ARRAYS) BIND(C)
  USE, INTRINSIC :: ISO_C_BINDING
  IMPLICIT NONE
  INTEGER (C_LONG), VALUE                :: ALPHA
  REAL (C_DOUBLE), INTENT(INOUT)         :: BETA
  INTEGER (C_LONG), INTENT(OUT)           :: GAMMA
  REAL (C_DOUBLE), DIMENSION(*), INTENT(IN) :: DELTA
  TYPE, BIND(C) :: PASS
    INTEGER (C_INT) :: LENC, LENF
    TYPE (C_PTR)    :: C, F
  END TYPE PASS
  TYPE (PASS), INTENT(INOUT) :: ARRAYS
  REAL (C_FLOAT), ALLOCATABLE, TARGET, SAVE :: ETA(:)
  REAL (C_FLOAT), POINTER :: C_ARRAY(:)
  ...
  ! Associate C_ARRAY with an array allocated in C
  CALL C_F_POINTER (ARRAYS%C, C_ARRAY, [ ARRAYS%LENC ])
  ...
  ! Allocate an array and make it available in C
  ARRAYS%LENF = 100
  ALLOCATE (ETA(ARRAYS%LENF))
  ARRAYS%F = C_LOC(ETA)
  ...
END SUBROUTINE SIMULATION

```

C Structure 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 of type REAL (C_DOUBLE), which has 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.13.4 Example of calling C functions with noninteroperable data (18.10)

- 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.13.5 Example of opaque communication between C and Fortran (18.3)

- 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
```



```

1      SUBROUTINE RANDOM_UNIFORM(STREAM, NUMBER)
2          IMPORT :: RANDOM_STREAM, C_DOUBLE
3          CLASS(RANDOM_STREAM), INTENT(INOUT) :: STREAM
4          REAL(C_DOUBLE), INTENT(OUT) :: NUMBER
5      END SUBROUTINE RANDOM_UNIFORM
6  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, non-parameterized derived type:

```

10     TYPE :: URNG_STATE ! No BIND(C), as this type is not interoperable
11     CLASS(RANDOM_STREAM), ALLOCATABLE :: STREAM
12 END TYPE URNG_STATE

```

- 3 The following two procedures will enable a C program to use our Fortran uniform random number generator:

```

15     ! Initialize a uniform random number generator:
16     SUBROUTINE INITIALIZE_URNG(STATE_HANDLE, METHOD) &
17         BIND(C, NAME="InitializeURNG")
18         TYPE(C_PTR), INTENT(OUT) :: STATE_HANDLE
19         ! An opaque handle for the URNG
20         CHARACTER(C_CHAR), DIMENSION(*), INTENT(IN) :: METHOD
21         ! The algorithm to be used
22
23         TYPE(URNG_STATE), POINTER :: STATE
24         ! An actual URNG object
25
26         ALLOCATE(STATE)
27         ! There needs to be a corresponding finalization
28         ! procedure to avoid memory leaks, not shown in this example
29         ! Allocate STATE%STREAM with a dynamic type depending on METHOD
30         ...
31         STATE_HANDLE=C_LOC(STATE)
32         ! Obtain an opaque handle to return to C
33     END SUBROUTINE INITIALIZE_URNG
34
35     ! Generate a random number:
36     SUBROUTINE GENERATE_UNIFORM(STATE_HANDLE, NUMBER) &
37         BIND(C, NAME="GenerateUniform")
38         TYPE(C_PTR), INTENT(IN), VALUE :: STATE_HANDLE
39         ! An opaque handle: Obtained via a call to INITIALIZE_URNG
40         REAL(C_DOUBLE), INTENT(OUT) :: NUMBER
41
42         TYPE(URNG_STATE), POINTER :: STATE
43         ! A pointer to the actual URNG
44
45         CALL C_F_POINTER(CPTR=STATE_HANDLE, FPTR=STATE)
46         ! Convert the opaque handle into a usable pointer
47         CALL STATE%STREAM%NEXT(NUMBER)
48         ! Use the type-bound procedure NEXT to generate NUMBER
49     END SUBROUTINE GENERATE_UNIFORM

```

C.13.6 Using assumed type to interoperate with C

C.13.6.1 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.13.6.2 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)
```

- 1 4 In those invocations, x and y are passed directly with sequence association, but it is necessary to make an
2 array expression containing the value of z to pass it.
- 3 5 The second method provides a Fortran interface which is easier to use, but requires writing a separate C
4 wrapper routine. With this method, a **C descriptor** is created because the buffer is **assumed-rank** in the
5 Fortran interface; the use of an optional argument is also demonstrated.

```

6      INTERFACE
7          SUBROUTINE example_send(buffer, handle, status) BIND(C, NAME="EG_send_fortran")
8              USE, INTRINSIC :: ISO_C_BINDING
9              TYPE(*), CONTIGUOUS, INTENT (IN) :: buffer(..)
10             INTEGER (C_INT), INTENT (IN) :: handle
11             INTEGER (C_INT), INTENT(OUT), OPTIONAL :: status
12         END SUBROUTINE
13     END INTERFACE

```

- 14 6 It is assumed that this interface is declared in the specification part of a module MOD_EXAMPLE_NEW.
15 Example invocations from Fortran are then

```

16      USE, INTRINSIC :: iso_c_binding
17      USE mod_example_new
18
19      TYPE, BIND(C) :: my_derived
20          INTEGER(C_INT) :: len_used
21          REAL(C_FLOAT) :: stuff(100)
22      END TYPE
23      TYPE(my_derived) :: w(3)
24      REAL(C_FLOAT) :: x(100)
25      INTEGER(C_INT) :: y(10,10)
26      REAL(C_DOUBLE) :: z
27      INTEGER(C_INT) :: status, handle
28      ...
29      ! Assign values to w, x, y, z and initialize handle.
30      ...
31      ! Send values in w, x, y, and z using example_send.
32      CALL example_send(w, handle, status)
33      CALL example_send(x, handle)
34      CALL example_send(y, handle)
35      CALL example_send(z, handle)
36      CALL example_send(y(:,5), handle) ! Fifth column of y.
37      CALL example_send(y(1,5), handle) ! Scalar y(1,5) passed by descriptor.

```

- 38 7 The wrapper routine can be written in C as follows.

```

39      #include "ISO_Fortran_binding.h"
40
41      void EG_send_fortran(const CFI_cdesc_t *buffer, const HANDLE_t *handle, int *status)
42      {
43          int status_local;
44          size_t buffer_size;
45          int i;
46
47          buffer_size = buffer->elem_len;
48          for (i=0; i<buffer->rank; i++) {
49              buffer_size *= buffer->dim[i].extent;
50          }
51          status_local = EXAMPLE_send(buffer->base_addr, buffer_size, handle);
52          if (status != NULL) *status = status_local;
53      }

```

1 C.13.7 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.

```

7      SUBROUTINE SIMPLE_SEND(buffer, nbytes)
8          USE comm_info, ONLY: my_node, r_node, r_addr
9          USE, INTRINSIC :: ISO_C_BINDING
10         IMPLICIT NONE
11
12         TYPE(*), INTENT (IN) :: buffer(*)
13         INTEGER                :: nbytes, ierr
14
15         INTERFACE
16             SUBROUTINE actual_Send(buffer, nbytes, node, addr, ierr) &
17             BIND(C, NAME="ACTUAL_Send")
18                 IMPORT :: C_SIZE_T, C_INT, C_INTPTR_T
19                 TYPE(*), INTENT (IN) :: buffer(*)
20                 INTEGER(C_SIZE_T), VALUE :: nbytes
21                 INTEGER(C_INT), VALUE :: node
22                 INTEGER(C_INTPTR_T), VALUE :: addr
23                 INTEGER(C_INT), INTENT(OUT) :: ierr
24             END SUBROUTINE actual_Send
25         END INTERFACE
26
27         CALL actual_Send(buffer, INT(nbytes, C_SIZE_T), r_node, r_addr, ierr)
28
29         IF (ierr /= 0) THEN
30             PRINT *, "Error sending from node", my_node, "to node", r_node
31             PRINT *, "Program Aborting" ! Or call a recovery procedure
32             ERROR STOP ! Omit in the recovery case
33         END IF
34     END SUBROUTINE simple_Send

```

35 C.13.8 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:

```

42     INTERFACE ieee_support_divide
43         MODULE PROCEDURE ieee_support_divide_noarg, ieee_support_divide_onearg_r, &
44             ieee_support_divide_onearg_d
45     END INTERFACE ieee_support_divide
46
47     ...
48
49     LOGICAL FUNCTION ieee_support_divide_noarg ()
50         ieee_support_divide_noarg = .TRUE.
51     END FUNCTION ieee_support_divide_noarg
52

```

```

1      LOGICAL FUNCTION ieee_support_divide_onearg_r (x)
2          REAL, INTENT (IN) :: x(..)
3          ieee_support_divide_onearg_r4 = .TRUE.
4      END FUNCTION ieee_support_divide_onearg_r
5
6      LOGICAL FUNCTION ieee_support_divide_onearg_d (x)
7          DOUBLE PRECISION, INTENT (IN) :: x(..)
8          ieee_support_divide_onearg_r8 = .TRUE.
9      END FUNCTION ieee_support_divide_onearg_d

```

C.13.9 Processing assumed-rank in C

1 The example shown below calculates the product of individual elements of arrays B and C and returns the result in array A. 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 This demonstrates runtime error detection even though these specific errors could have been detected at compile-time, if the interface declared the arrays as “`INTEGER (C_INT), DIMENSION (:, :)`”.

3 The Fortran interface is:

```

19      INTERFACE
20          FUNCTION elemental_mult(a, b, c) BIND(C, NAME="elemental_mult_c") RESULT(err)
21              USE, INTRINSIC :: ISO_C_BINDING
22              INTEGER(C_INT) :: err
23              TYPE(*), DIMENSION(..) :: a, b, c
24          END FUNCTION elemental_mult
25      END INTERFACE

```

4 The definition of the C function is:

```

27      #include "ISO_Fortran_binding.h"
28
29      int elemental_mult_c(CFI_cdesc_t *a_desc, CFI_cdesc_t *b_desc, CFI_cdesc_t *c_desc)
30      {
31          size_t i, j, ni, nj;
32          int err = 1; /* this error code represents all errors */
33          char * a_col = (char*) a_desc->base_addr;
34          char * b_col = (char*) b_desc->base_addr;
35          char * c_col = (char*) c_desc->base_addr;
36          char *a_elt, *b_elt, *c_elt;
37
38          /* Only support int. */
39          if (a_desc->type != CFI_type_int || b_desc->type != CFI_type_int ||
40              c_desc->type != CFI_type_int) {
41              return err;
42          }
43          /* Only support two dimensions. */
44          if (a_desc->rank != 2 || b_desc->rank != 2 || c_desc->rank != 2) {
45              return err;
46          }
47
48          ni = a_desc->dim[0].extent;
49          nj = a_desc->dim[1].extent;
50
51          /* Ensure the shapes conform. */
52          if (ni != b_desc->dim[0].extent || ni != c_desc->dim[0].extent) return err;

```

```

1      if (nj != b_desc->dim[1].extent || nj != c_desc->dim[1].extent) return err;
2
3      /* Multiply the elements of the two arrays. */
4      for (j = 0; j < nj; j++) {
5          a_elt = a_col;
6          b_elt = b_col;
7          c_elt = c_col;
8          for (i = 0; i < ni; i++) {
9              *(int*)a_elt = *(int*)b_elt * *(int*)c_elt;
10             a_elt += a_desc->dim[0].sm;
11             b_elt += b_desc->dim[0].sm;
12             c_elt += c_desc->dim[0].sm;
13         }
14         a_col += a_desc->dim[1].sm;
15         b_col += b_desc->dim[1].sm;
16         c_col += c_desc->dim[1].sm;
17     }
18     return 0;
19 }

```

20 C.13.10 Creating a contiguous copy of an array

21 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.

25 2 The C functions are:

```

26     #include "ISO_Fortran_binding.h"
27     /* Other necessary includes omitted. */
28
29     /*
30      * Returns the number of elements in the object described by desc.
31      * If it is an array, it need not be contiguous.
32      * (The number of elements could be zero).
33      */
34     size_t numElements(const CFI_cdesc_t * desc)
35     {
36         CFI_rank_t r;
37         size_t num = 1;
38
39         for (r = 0; r < desc->rank; r++) {
40             num *= desc->dim[r].extent;
41         }
42         return num;
43     }
44
45     /*
46      * Auxiliary recursive function to copy an array of a given rank.
47      * Recursion is useful because an array of rank n is composed of an
48      * ordered set of arrays of rank n-1.
49      */
50     static void *_copyToContiguous (const CFI_cdesc_t *vald, void *output,
51                                     const void *input, CFI_rank_t rank)
52     {
53         CFI_index_t e;
54
55         if (rank == 0) {
56             /* Copy scalar element. */

```

```

1      memcpy (output, input, vald->elem_len);
2      output = (void *) ((char *)output + vald->elem_len);
3  }
4  else {
5      for (e = 0; e < vald->dim[rank-1].extent; e++) {
6          /* Recurse on subarrays of lesser rank. */
7          output = _copyToContiguous (vald, output, input, rank-1);
8          input = (void *) ((char *)input + vald->dim[rank].sm);
9      }
10 }
11 return output;
12 }
13
14 /*
15  * General routine to copy the elements in the array described by vald
16  * to buffer, as done by sequence association. The array itself can
17  * be non-contiguous. This is not the most efficient approach.
18  */
19 void copyToContiguous (void * buffer, const CFI_cdesc_t * vald) {
20     _copyToContiguous (vald, buffer, vald->base_addr, vald->rank);
21 }

```

22 C.13.11 Changing the attributes of an array

- 23 1 A C programmer might want to call more than one Fortran procedure and the attributes of an array involved
24 might differ between the procedures. In this case, it is necessary to set up more than one **C descriptor** for
25 the array. For example, this code fragment initializes the first **C descriptor** for an allocatable entity of rank
26 2, calls a procedure that allocates the array described by the first **C descriptor**, constructs the second **C**
27 **descriptor** by invoking **CFI_establish** with the value **CFI_attribute_other** for the **attribute** parameter, then
28 calls a procedure that expects an **assumed-shape array**.

```

29 CFI_CDESC_T(2) loc_alloc, loc_assum;
30 CFI_cdesc_t * desc_alloc = (CFI_cdesc_t *)&loc_alloc,
31      * desc_assum = (CFI_cdesc_t *)&loc_assum;
32 CFI_index_t extents[2];
33 CFI_rank_t rank = 2;
34 int flag;
35
36 flag = CFI_establish(desc_alloc,
37      NULL,
38      CFI_attribute_allocatable,
39      CFI_type_double,
40      sizeof(double),
41      rank,
42      NULL);
43
44 Fortran_factor (desc_alloc, ...); /* Allocates array described by desc_alloc. */
45
46 /* Extract extents from descriptor. */
47 extents[0] = desc_alloc->dim[0].extent;
48 extents[1] = desc_alloc->dim[1].extent;
49
50 flag = CFI_establish(desc_assum,
51      desc_alloc->base_addr,
52      CFI_attribute_other,
53      CFI_type_double,
54      sizeof(double),
55      rank,
56      extents);

```



```

1
2 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 zero even if the corresponding entries in `desc_alloc` have different values.

C.13.12 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. It 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.

```

10 SUBROUTINE set_all(int_array, val) BIND(C)
11     INTEGER(C_INT) :: int_array(:)
12     INTEGER(C_INT), VALUE :: val
13     int_array = val
14 END SUBROUTINE
15
16 INTERFACE
17     SUBROUTINE set_odd(int_array, val) BIND(C)
18         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_INT
19         INTEGER(C_INT) :: int_array(:)
20         INTEGER(C_INT), VALUE :: val
21     END SUBROUTINE
22 END INTERFACE
23
24 #include "ISO_Fortran_binding.h"
25
26 void set_odd(CFI_cdesc_t *int_array, int val)
27 {
28     CFI_index_t lower_bound[1], upper_bound[1], stride[1];
29     CFI_CDESC_T(1) array;
30     int status;
31     /* Create a new descriptor which will contain the section. */
32     status = CFI_establish((CFI_cdesc_t *)&array,
33                           NULL,
34                           CFI_attribute_other,
35                           int_array->type,
36                           int_array->elem_len,
37                           /* rank */ 1,
38                           /* extents is ignored *//NULL);
39
40     lower_bound[0] = int_array->dim[0].lower_bound;
41     upper_bound[0] = lower_bound[0] + (int_array->dim[0].extent - 1);
42     stride[0] = 2;
43
44     status = CFI_section((CFI_cdesc_t *)&array,
45                         int_array,
46                         lower_bound,
47                         upper_bound,
48                         stride);
49
50     set_all( (CFI_cdesc_t *) &array, val);
51
52     /* Here one could make use of int_array and access all its data. */
53 }

```

- 2 The `set_odd` procedure can be called from Fortran as follows:

```

1      INTEGER(C_INT) :: d(5)
2      d = (/ 1, 2, 3, 4, 5 /)
3      CALL set_odd(d, -1)
4      PRINT *, d

```

5 3 This program will print something like:

```

6      -1      2      -1      4      -1

```

7 4 During execution of the subroutine SET_ALL, its dummy argument INT_ARRAY would have size (and upper bound) 3.

9 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.

12 6 Following is an example C function that dynamically generates a **C descriptor** for an **assumed-shape array** and calls `set_odd`.

```

14      #include <stdio.h>
15      #include <stdlib.h>
16      #include "ISO_Fortran_binding.h"
17
18      #define ARRAY_SIZE 5
19
20      void example_of_calling_set_odd(void)
21      {
22          CFI_CDESC_T(1) d;
23          CFI_index_t extent[1];
24          CFI_index_t subscripts[1];
25          void *base;
26          int i, status;
27          base = malloc(ARRAY_SIZE*sizeof(int));
28          extent[0] = ARRAY_SIZE;
29          status = CFI_establish((CFI_cdesc_t *)&d,
30                                base,
31                                CFI_attribute_other,
32                                CFI_type_int,
33                                /* element length is ignored */ 0,
34                                /* rank */ 1,
35                                extent);
36          set_odd((CFI_cdesc_t *)&d, -1);
37          for (i=0; i<ARRAY_SIZE; i++) {
38              subscripts[0] = i;
39              printf(" %d", *((int *)CFI_address((CFI_cdesc_t *)&d, subscripts)));
40          }
41          putc('\n', stdout);
42          free(base);
43      }

```

44 The above C function will print similar output to that of the preceding Fortran program.

45 C.13.13 Use of CFI_setpointer

46 1 The C function `change_target` modifies a pointer to an integer variable to become associated with a global variable defined inside C:

```

48      #include "ISO_Fortran_binding.h"

```

```

1
2     int y = 2;
3
4     void change_target(CFI_cdesc_t *ip) {
5         CFI_CDESC_T(0) yp;
6         int status;
7         /* Make local yp point at y. */
8         status = CFI_establish((CFI_cdesc_t *)&yp,
9                               &y,
10                              CFI_attribute_pointer,
11                              CFI_type_int,
12                              /* elem_len is ignored */ sizeof(int),
13                              /* rank */ 0,
14                              /* extents are ignored */ NULL);
15         /* Pointer-associate ip with (the target of) yp. */
16         status = CFI_setpointer(ip, (CFI_cdesc_t *)&yp, NULL);
17         if (status != CFI_SUCCESS) {
18             ... Report run time error.
19         }
20     }

```

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.13.14 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.13.6.2. 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

```

1      void MPI_Send_f(CFI_cdesc_t *buf, int count, MPI_Datatype_f datatype, int dest,
2                      int tag, MPI_Datatype_f comm, int *ierror);

```

3 3 This wrapper function will convert MPI_Datatype_f and MPI_Comm_f to MPI_Datatype and MPI_Comm, and
 4 produce a contiguous void * buffer from CFI_cdesc_t *buf (if necessary).

5 4 Similarly, the wrapper function for MPI_Comm_set_name could have the C prototype:

```

6      void MPI_Comm_set_name_f(MPI_Comm comm, CFI_cdesc_t *comm_name, int *ierror);

```

7 5 The Fortran handle types and interfaces are defined in the module MPI_F08. For example,

```

8      MODULE mpi_f08
9      ...
10     TYPE, BIND(C) :: mpi_comm
11     PRIVATE
12     INTEGER(C_INT) :: mpi_val
13     END TYPE mpi_comm
14
15     INTERFACE
16     SUBROUTINE MPI_SEND(buf, count, datatype, dest, tag, comm, ierror) &
17     BIND(C, NAME='MPI_Send_f')
18     USE, INTRINSIC :: ISO_C_BINDING
19     IMPORT :: MPI_Datatype, MPI_Comm
20     TYPE(*), DIMENSION(..), INTENT (IN) :: buf
21     INTEGER(C_INT), VALUE, INTENT (IN) :: count, dest, tag
22     TYPE(mpi_datatype), INTENT (IN) :: datatype
23     TYPE(mpi_comm), INTENT (IN) :: comm
24     INTEGER(C_INT), OPTIONAL, INTENT (OUT) :: ierror
25     END SUBROUTINE mpi_send
26
27     SUBROUTINE mpi_comm_set_name(comm, comm_name, ierror) &
28     BIND(C, NAME='MPI_Comm_set_name_f')
29     USE, INTRINSIC :: ISO_C_BINDING
30     IMPORT :: mpi_comm
31     TYPE(mpi_comm), INTENT (IN) :: comm
32     CHARACTER(KIND=C_CHAR, LEN=*), INTENT (IN) :: comm_name
33     INTEGER(C_INT), OPTIONAL, INTENT (OUT) :: ierror
34     END SUBROUTINE mpi_comm_set_name
35     END INTERFACE
36     ...
37     END MODULE mpi_f08

```

38 6 Some examples of invocation from Fortran are:

```

39     USE, INTRINSIC :: ISO_C_BINDING
40     USE :: MPI_f08
41
42     TYPE(mpi_comm) :: comm
43     REAL :: x(100)
44     INTEGER :: y(10,10)
45     REAL(KIND(1.0d0)) :: z
46     INTEGER :: dest, tag, ierror
47     ...
48     ! Assign values to x, y, z and initialize MPI variables.
49     ...
50
51     ! Set the name of the communicator.
52     CALL mpi_comm_set_name(comm, "Communicator Name", ierror)

```

```

1
2      ! Send values in x, y, and z.
3      CALL mpi_send(x, 100, MPI_REAL, dest, tag, comm, ierror)
4      IF (ierror/=0) PRINT *, 'WARNING: X send error', ierror
5      CALL mpi_send(y(3,:), 10, MPI_INTEGER, dest, tag, comm)
6      CALL mpi_send(z, 1, MPI_DOUBLE_PRECISION, dest, tag, comm)

```

7 7 The first example sends the entire array X and includes the optional error argument return value. The
8 second example sends a noncontiguous subarray (the third row of Y) and the third example sends a scalar
9 Z. Note the differences between the calls in this example and those in [C.13.6.2](#).

10 C.14 Clause 19 notes

11 C.14.1 Examples of global identifiers and binding labels ([19.2](#))

12 Example 1:

```

13      MODULE M1
14      INTERFACE
15          SUBROUTINE S() BIND(C,NAME='X')
16      END
17      END INTERFACE
18  END MODULE
19  MODULE M2
20      INTERFACE
21          SUBROUTINE S() BIND(C,NAME='Y')
22      END
23      END INTERFACE
24  END MODULE

```

25 1 The name S in each module is a local identifier. The two interfaces declare two different external proced-
26 ures, one with the global identifier “X”, the other with the global identifier “Y”.

27 Example 2:

```

28      MODULE M1
29      INTERFACE
30          SUBROUTINE S1() BIND(C,NAME='X')
31      END
32      END INTERFACE
33  END MODULE
34  MODULE M2
35      INTERFACE
36          SUBROUTINE S2() BIND(C,NAME='X')
37      END
38      END INTERFACE
39  END MODULE

```

40 2 The names S1 and S2 are local identifiers. The interfaces declare the same external procedure, which has
41 the global identifier “X”.

42 C.14.2 Examples of host association ([19.5.1.4](#))

43 1 The first two examples are examples of valid [host association](#). The third example is an example of invalid
44 [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 (10.1.12).

Index

In the index, entries in *italics* denote BNF terms, and page numbers in **bold face** denote primary text or definitions.

Symbols

—, 174
 <, 178, 502
 <=, 178
 >, 178
 >=, 178
 *, 64, 67, 69, 70, 76, 116, 120, 131, 157, 174, 269, 300, 315, 321, 345, 366
 **, 174
 +, 174
 .. assumed-rank specifier, 121
 .AND., 169, 170, 173, 177, **177**, 391
 .EQ., 168, 170, 173, 178, **178**, 179, 180, 335, 502
 .EQV., 169, 170, 173, 177, **177**
 .FALSE., **79**, 538
 .GE., 168, 170, 173, 178, **178**, 180, 335, 502
 .GT., 168, 170, 173, 178, **178**, 180, 335, 502
 .LE., 168, 170, 173, 178, **178**, 180, 335, 502
 .LT., 168, 170, 173, 178, **178**, 180, 335, 502
 .NE., 168, 170, 173, 178, **178**, 179, 180, 335, 502
 .NEQV., 169, 170, 173, 177, **177**, 462
 .NIL., 59, 345, **345**, 347, 348
 .NOT., 169, 170, 173, 177, **177**
 .OR., 169, 170, 173, 177, **177**, 392
 .TRUE., **79**, 538
 /, 174
 / edit descriptor, 312
 //, 176
 /=, 178, 502
 : edit descriptor, 312
 ;, 63
 <=, 502
 ==, 178, 502
 >, 502
 >=, 502
 &, 63, 320

A

A edit descriptor, 309
 ABS, **388**, 505
 ABSTRACT, 80, **80**, 81, 96, **334**, 335
 ABSTRACT attribute, 80, **96**
 abstract interface, **17**, 327, **333**, 336, 342, 364, 572, 576
 abstract interface block, **18**, 335, 336
 abstract type, **29**, 68, 69, 93, **96**, 97, 99, 148, 157
ac-do-variable (R785), 108, **108**, 184, 187, 574, 575
ac-implied-do (R783), 107, **107**, 108, 172, 575
ac-implied-do-control (R784), 107, **108**, 172, 184, 186, 187, 575
ac-spec (R779), 107, **107**
ac-value (R782), 107, **107**, 108
access-id (R832), 128, **128**, 129
access-name, 128, 129
access-spec (R807), 80, 85, 86, 91–94, 105, 110, 113, **113**, 128, 129, 337, 341
access-stmt (R831), 45, 105, 113, 128, **128**, 129
 ACCESS= specifier, 253, **254**, 283, **284**
 accessibility attribute, **113**, 129, 327
 accessibility statement, 128
 ACHAR, 79, 191, **388**
 ACOS, **389**
 ACOSD, **389**
 ACOSH, 38, **389**
 ACOSPI, **390**
 ACQUIRED_LOCK= specifier, **239**, 588, 591
 action, 244
action-stmt (R515), 45, **45**, 172, 220, 228
 ACTION= specifier, 254, **255**, 283, **284**, 627
 active image, **16**, 50, 54, 159, 163, 164, 208, 209, 238, 241, 380, 381, 455
 actual argument, **3**, 37, 39, 50, 54–56, 70, 73, 83, 95, 96, 118, 120–124, 148, 150, 160, 163, 172, 182, 217, 275, 338, 339, 344–357, 359, 360, 369, 370, 372, 374–377, 381, 386, 388,

- 414, 441, 459, 470, 487, 489, 490, 494, 506, 543–546, 548, 551–553, 576, 580, 582, 585, 589–591, 600, 644, 645, 647, 649
- actual-arg* (R1524), 344, **344**, 345, 347
- actual-arg-spec* (R1523), 100, 344, **344**, 345
- add-op* (R1010), 60, 167, 168, **168**
- add-operand* (R1006), 167, **167**, 168, 171
- ADJUSTL, **390**
- ADJUSTR, **390**
- ADVANCE= specifier, 260, 261, **262**, 274, 626
- advancing input/output statement, 247
- AIMAG, 149, **390**
- AINTE, **391**
- ALL, **138**, **391**
- alloc-opt* (R930), 156, **156**, 158, 164
- allocatable, **3**, 38, 55, 56, 67, 70, 71, 82, 87, 88, 90, 95, 98, 100, 101, 111, 115, 116, 120, 122, 123, 128, 131, 141, 143, 148, 150, 156, 157, 160–164, 183, 186, 188–191, 194, 217, 232, 266, 267, 272, 317, 332–334, 344, 345, 349, 352, 356, 365, 372, 376, 392, 413, 424, 441, 442, 454–456, 459, 471, 475, 478, 481, 485, 489, 490, 500, 506, 538, 545, 548–550, 555, 562, 564–566, 579, 580, 589
- ALLOCATABLE attribute, 67, 69, 70, 79, 85, 113, **113**, 115, 116, 120, 121, 125, 127, 129, 148, 152, 205, 212, 216, 223, 224, 330, 334, 339, 340, 346, 348, 352, 356, 364, 374, 550, 579, 585, 586, 646, 650
- ALLOCATABLE statement, **129**
- allocatable-decl* (R834), 129, **129**
- allocatable-stmt* (R833), 45, **129**, 577
- ALLOCATE statement, 67, 69, 76, 77, 117, 120, **156**, 160, 164, 165, 194, 231, 494, 497, 559, 560, 580, 581, 587, 589, 593
- allocate-coarray-spec* (R940), 157, **157**
- allocate-coshape-spec* (R941), 157, **157**
- allocate-object* (R934), 76, 77, 157, **157**, 158–160, 162–165, 231, 494, 496, 497, 591, 593
- allocate-shape-spec* (R935), 157, **157**, 159, 160
- allocate-stmt* (R929), 45, **156**, 591
- ALLOCATED, 160, 165, 185, **392**
- allocation* (R933), 156, 157, **157**, 158–160
- allocation status, **56**, 98, 100, 101, 123, 126, 128, 160, **160**, 161–164, 217, 232, 235, 352, 357, 392, 455, 459, 566, 585, 589
- alphanumeric-character* (R601), 58, **58**, 59
- alt-return-spec* (R1525), 228, 344, 345, **345**
- alternate return indicator, **369**
- ancestor component, **97**
- ancestor-module-name*, 329
- and-op* (R1020), 60, 169, **169**
- and-operand* (R1015), 169, **169**
- ANINT, **392**
- ANY, **392**
- arg-name*, 86, 88, 93
- argument
 - dummy, 350
- argument association, **4**, 67, 77, 87, 88, 116, 120, 127, 128, 162, 164, 331, 347–349, 359, 367, 555, 576, 582, 584, 585, 599, 645
- argument keyword, **19**, 56, 333, 337, 347, 376, 381, 506, 572, **574**, 635
- arithmetic IF statement, **599**
- array, **3**, 55, 118–121, 150–154
 - assumed-shape, **4**, 70, 116–119, 127, 140, 155, 334, 349–353, 357, 364, 538, 551, 566, 644, 659, 664, 666
 - assumed-size, **4**, 118, 120–122, 127, 140, 150, 151, 166, 184, 187, 212, 224, 265, 349, 350, 352, 355, 356, 441, 475, 478, 489, 543, 546, 550, 552, 553, 555, 559, 562–566, 655, 657, 659
 - deferred-shape, **4**, 120, 127
 - explicit-shape, **4**, 70, 87, 115, 118, 119, 187, 350, 352, 355, 550, 552, 553
- array bound, **7**, 86, 87, 89, 112, 185
- array constructor, 66, 107, **107**
- array element, **3**, 55, 151
- array element order, 152–153
- array pointer, **3**, 116, 117, 120, 183, 395, 550
- array section, **3**, 117, 130, 132, 149, 151–155, 205, 249, 250, 350, 351, 357, 579, 581
- array-constructor* (R778), **107**, 108, 166, 167
- array-element* (R917), 131, 141, 146, 147, **150**, 216
- array-name*, 133, 577
- array-section* (R918), 146, 151, **151**, 153, 216
- array-spec* (R814), 34, 110, 111, 113, 118, **118**, 120, 121, 129, 133, 135, 143, 144
- ASCII character, **4**, **75**, **79**, 188, 249, 250, 267, 300, 315, 316, 389, 408, 431, 435, 443, 444, 457, 472
- ASCII collating sequence, **79**, 388, 389, 408, 431, 435, 443, 444, 457
- ASIN, **393**
- ASIND, **393**
- ASINH, **393**
- ASINPI, **394**

- ASSIGN statement, **598**
- assigned format, **598**
- assigned GO TO statement, **598**
- ASSIGNMENT, **93, 192, 335, 339, 340**
- assignment, **187–202**
 - defined, **92, 192, 339**
 - elemental, **14, 192**
 - elemental array (FORALL), **199**
 - masked array (WHERE), **197**
 - pointer, **192**
- assignment statement, **38, 53, 67, 95, 187, 201, 231, 232, 496, 536, 586, 588**
- assignment-stmt* (R1033), **45, 187, 188, 197, 200, 591**
- ASSOCIATE construct, **55, 204, 207, 357, 574, 575, 579, 591**
- associate name, **4, 67, 70, 99, 115, 121, 162, 204, 205, 208, 226, 575, 579, 585, 591**
- ASSOCIATE statement, **55, 204, 579**
- associate-construct* (R1102), **45, 204, 204**
- associate-construct-name*, **204**
- associate-name*, **204, 223, 225–227, 574**
- associate-stmt* (R1103), **204, 204, 228**
- ASSOCIATED, **37–39, 161, 165, 185, 377, 394**
- associating entity, **4, 55, 77, 156, 204, 205, 208, 209, 227, 367, 584, 585**
- association, **4**
 - argument, **4, 67, 77, 87, 88, 116, 120, 127, 128, 162, 164, 331, 347–349, 359, 367, 555, 576, 582, 584, 585, 599, 645**
 - common, **144**
 - construct, **5, 162, 164, 576, 579, 582, 584, 585**
 - equivalence, **142**
 - host, **5, 47, 69, 70, 77, 114, 128, 131, 136, 144, 184, 185, 194, 329, 331, 356, 370, 372–374, 574, 576–579, 581, 582, 584, 585, 669**
 - inheritance, **5, 57, 97, 99, 582, 584**
 - linkage, **5, 568, 576, 579, 579**
 - name, **5, 56, 576, 582**
 - pointer, **5, 53, 57, 96, 98, 101, 117, 123, 125, 127, 128, 148, 162, 164, 192, 194, 195, 216, 217, 232, 235, 268, 332, 348, 350, 353, 355, 356, 365, 367, 382, 394, 395, 455, 459, 541–543, 555, 566, 570, 579–644**
 - sequence, **355**
 - storage, **5, 56, 57, 141–143, 368, 372, 481, 582–584**
 - use, **5, 38, 47, 57, 70, 77, 97, 113, 114, 125, 128, 136, 140, 142, 143, 184, 185, 194, 326, 325–329, 336, 367, 372, 374, 574, 576, 580**
- association* (R1104), **204, 204**
- association status, *see* pointer association status
- assumed character length function, **77**
- assumed type parameter, **30, 67, 69, 349, 352**
- assumed-implied-spec* (R823), **120, 120, 121**
- assumed-rank dummy data object, **6, 54, 70, 94, 95, 116–118, 155, 223, 224, 332, 334, 340, 345, 349–352, 357, 364, 394, 440, 441, 459, 467, 475, 478, 489, 538, 546, 551, 659–661**
- assumed-rank-spec* (R828), **118, 121**
- assumed-shape array, **4, 70, 116–119, 127, 140, 155, 334, 349–353, 357, 364, 538, 551, 566, 644, 659, 664, 666**
- assumed-shape-bounds-spec* (R821), **118, 119, 119**
- assumed-shape-spec* (R820), **118, 119, 119**
- assumed-size array, **4, 118, 120–122, 127, 140, 150, 151, 166, 184, 187, 212, 224, 265, 349, 350, 352, 355, 356, 441, 475, 478, 489, 543, 546, 550, 552, 553, 555, 559, 562–566, 655, 657, 659**
- assumed-size-spec* (R824), **118, 120, 120**
- assumed-type, **6, 69, 70, 349, 350, 364, 545, 551, 661**
- ASYNCHRONOUS attribute, **114, 114, 130, 205, 212, 216, 263, 327, 328, 332, 334, 351, 410, 468, 566, 567, 570, 576, 577**
- asynchronous communication, **114, 570**
- asynchronous input/output, **114, 253, 255, 258, 262, 263, 265, 268, 269, 276, 278–280, 282, 285, 287, 288**
- ASYNCHRONOUS statement, **130, 206, 330, 574, 577**
- asynchronous-stmt* (R835), **45, 130**
- ASYNCHRONOUS= specifier, **254, 255, 260, 261, 262, 283, 285**
- AT edit descriptor, **309**
- ATAN, **395**
- ATAN2, **40, 395**
- ATAN2D, **396**
- ATAN2PI, **396**
- ATAND, **397**
- ATANH, **397**
- ATANPI, **397**
- atomic subroutine, **27, 50, 232, 233, 376, 380, 382, 398–402, 422, 492, 497, 590**
- ATOMIC_ADD, **398, 493, 496**
- ATOMIC_AND, **398**
- ATOMIC_CAS, **399**

- ATOMIC_DEFINE, **399**, 651
 ATOMIC_FETCH_ADD, **400**
 ATOMIC_FETCH_AND, **400**
 ATOMIC_FETCH_OR, **401**
 ATOMIC_FETCH_XOR, **401**
 ATOMIC_INT_KIND, 398–403, **492**
 ATOMIC_LOGICAL_KIND, 399, 402, **492**
 ATOMIC_OR, **402**
 ATOMIC_REF, **402**, 651
 ATOMIC_XOR, **402**
attr-spec (R802), 110, **110**, 111, 113, 135
 attribute, **6**, 68, 79, 83, 110, 111, 113–128, 328
 ABSTRACT, **80**, **96**
 accessibility, **113**, 129, 327
 ALLOCATABLE, 67, 69, 70, 79, 85, 113, **113**, 115, 116, 120, 121, 125, 127, 129, 148, 152, 205, 212, 216, 223, 224, 330, 334, 339, 340, 346, 348, 352, 356, 364, 374, 550, 579, 585, 586, 646, 650
 ASYNCHRONOUS, 114, **114**, 130, 205, 212, 216, 263, 327, 328, 332, 334, 351, 410, 468, 566, 567, 570, 576, 577
 BIND, 5, 6, 52, 80–82, 96, **102**, 114, **114**, 128, 130, 141, 143, 193, 195, 216, 226, 330, 332, 334, 363, 365, 548, 549, 551, 566–569, 579, 586, 655
 CODIMENSION, 70, 86, 111, 115, **115**, 121, 130
 CONTIGUOUS, 85, 88, 116, **116**, 117, 118, 130, 154, 155, 195, 216, 332, 333, 350–353, 357, 551–553, 583
 DEFERRED, **92**, 93, 97
 DIMENSION, 86, 111, 118, **118**, 126, 133, 144
 EXTENDS, 96, **96**, 549
 EXTERNAL, 35, 36, 122, **122**, 125, 134, 136, 138, 194, 326, 330, 336, 341, 355, 360, 361, 576–578, 642
 INTENT, 122, **122**, 123, 124, 133, 216
 INTENT (IN), 122, **122**, 123, 124, 128, 212, 338–340, 349, 352, 353, 355, 357, 371–373, 377, 398–403, 409–411, 422, 423, 428, 429, 456, 465, 466, 508, 540–543, 566, 590, 644, 657
 INTENT (INOUT), 37, **122**, 123, 124, 127, 217, 339, 345, 351, 353, 360, 373–375, 388, 398–402, 408–411, 423, 428, 429, 454–456, 494, 496, 497, 591, 656
 INTENT (OUT), 37–39, 70, 94, 96, 120, 122, **122**, 123, 124, 127, 163, 184, 339, 345, 351, 353, 360, 371, 373–375, 388, 398–403, 409–411, 415, 416, 422, 423, 427–429, 455–457, 463, 466, 482, 483, 510–512, 540, 542, 543, 566, 580, 582, 586, 587, 589, 591, 657
 INTRINSIC, 122, 124, **124**, 125, 326, 343, 360, 361, 578
 NON_OVERRIDABLE, **92**, 93
 NON_RECURSIVE, 333, 363, **363**, 367, 369
 OPTIONAL, 69, 124, **124**, 128, 134, 184, 205, 212, 334
 PARAMETER, 53, 103, 112, 124, **124**, 125, 134, 147
 PASS, 86, **88**, 93, 344
 POINTER, 67, 69, 70, 79, 85, 86, 111, 120, 121, 125, **125**, 127, 132, 134, 148, 149, 152, 161, 193, 194, 205, 216, 223, 224, 332, 334–336, 339, 340, 342, 346, 348, 352, 354, 356–358, 364, 372, 374, 545, 550, 566, 579, 581, 585, 586, 608, 646, 650
 PRIVATE, 82, 98, 113, **113**, 129, 372, 633
 PROTECTED, 38, 125, **125**, 126, 135, 142, 216, 327
 PUBLIC, 98, 113, **113**, 129, 633
 SAVE, 41, 55, 89, 96, 112–115, 126, **126**, 130, 135, 142, 145, 163, 216, 343, 371, 581
 SEQUENCE, 80, **81**, 82, 83, 96, 143, 193, 195, 226, 548
 TARGET, 5, 38, 89, 125, 127, **127**, 135, 141, 145, 161, 162, 193, 205, 216, 224, 332, 334, 339, 349, 350, 352, 353, 357, 358, 410, 455, 468, 540, 543, 545, 566, 567, 580, 581, 590, 608, 644, 645
 VALUE, 70, 88, 94, 121, 127, **127**, 128, 135, 216, 268, 332, 334, 335, 338, 339, 348–351, 353, 364, 371, 374, 410, 468, 551, 552, 570, 581, 655, 656
 VOLATILE, 37, 39, 127, 128, **128**, 135, 193, 195, 205, 212, 216, 327, 328, 332, 334, 351, 353, 371, 576, 577, 582, 588, 590, 614
 attribute specification statements, 128–145
 automatic data object, **6**, 40, 41, 111, 112, 115, 126, 131, 141, 143, 587, 600

B

- B edit descriptor, 307
 BACKSPACE statement, 244, 248, 276, 279, **280**, 281, 626, 627
backspace-stmt (R1224), 45, **280**, 372
 base object, **6**, 114, 117, 141, **148**, **149**, 155, 184,

- 263, 356, 372, 374
- BESSEL_J0, **403**
- BESSEL_J1, **403**
- BESSEL_JN, **403**
- BESSEL_Y0, **404**
- BESSEL_Y1, **404**
- BESSEL_YN, **404**
- BGE, **405**
- BGT, **405**
- binary-constant* (R774), 106, **106**
- binary-reduce-op* (R1132), 211, **211**, 217
- BIND (C), *see* BIND attribute
- BIND attribute, 5, 6, 52, 80–82, 96, **102**, 114, **114**, 128, 130, 141, 143, 193, 195, 216, 226, **330**, 332, 334, 363, 365, 548, 549, 551, 566–569, 579, 586, 655
- BIND statement, **130**, 330, 567, 573
- bind-entity* (R837), 130, **130**
- bind-stmt* (R836), 45, **130**
- binding, **6**, 92, 93, **93**, 97, 98, 180, 192, 272, 278, 340, 362, 572–574
- binding label, **6**, 114, 333, 342, 363, 366, 567–569, 571, 572
- binding name, **6**, 92, **93**, 98, 344, 574
- binding-attr* (R752), 92, 93, **93**
- binding-name*, 92, 93, 344, 362, 574
- binding-private-stmt* (R747), 92, **92**, 94
- bit model, 378
- BIT_SIZE, 378, **406**, 456
- blank common, **8**, 111, 131, 143–145, 581, 584
- blank interpretation mode, 255
- blank-interp-edit-desc* (R1317), 296, **297**
- BLANK= specifier, 254, **255**, 260, 261, **263**, 276, 283, **285**, 314
- BLE, **406**
- block, **7**
 - interface, 327
- block* (R1101), **203**, 204, 206–209, 211, 213, 214, 216, 218–220, 223, 225
- BLOCK construct, 37, 38, 50, 52, 96, 112, 114, 116, 119, 126, 128, 136–140, 162, 184, **206**, 371, 574, 580, 581, 588, 590, 593
- block data program unit, **330**
- BLOCK DATA statement, 62, 325, **330**
- block scoping unit, **24**
- BLOCK statement, 112, 116, 119, **206**, 588
- block-construct* (R1107), 45, 206, **206**
- block-construct-name*, 206
- block-data* (R1420), 43, 138, 330, **330**
- block-data-name*, 330
- block-data-stmt* (R1421), 44, 330, **330**
- block-specification-part* (R1109), 206, **206**
- block-stmt* (R1108), 206, **206**, 228
- BLT, **406**
- BN edit descriptor, 314
- bound, **7**, 54–56, 85, 86, 98, 101, 119, 157, 158, 164, 165, 195, 232, 455, 575
- bounds, 118–121, 150–154
- bounds-remapping* (R1037), 193, **193**, 195
- bounds-spec* (R1036), 193, **193**, 195
- boz-literal-constant* (R773), 60, 102, 103, 106, **106**, 107, 108, 133, 188, 190, 267, 308, 378, 379, 405–408, 417, 419, 432, 435, 437, 438, 451, 467, 468
- branch, **228**, 369, 598
- branch target statement, **7**, 49, 61, 197, 215, 228, **228**, 229, 254, 259, 260, 279, 280, 282, 284, 345, 369
- BTEST, **407**
- BZ edit descriptor, 314
- C**
- C address, **7**, 540–546, 548, 549, 554, 559, 561, 588, 590, 657
- C descriptor, **7**, 163, 552–556, 558–567
- C_ALERT, **539**
- C_ASSOCIATED, **540**
- C_BACKSPACE, **539**
- C_BOOL, **538**, **539**
- C_CARRIAGE_RETURN, **539**
- C_CHAR, **539**, 543, 546, 608
- C_DOUBLE, **539**
- C_DOUBLE_COMPLEX, **539**
- C_F_POINTER, 540, **540**, 545
- C_F_PROCPONTER, 540, **542**, 544
- C_F_STRPONTER, 540, **543**
- C_FLOAT, **539**
- C_FLOAT_COMPLEX, **539**
- C_FORM_FEED, **539**
- C_FUNLOC, 372, 542, **544**, 569
- C_FUNPTR, 85, 96, 115, 148, 157, 158, 191, 538–540, 542, 544, **548**, 549, 590
- C_HORIZONTAL_TAB, **539**
- C_INT, **538**
- C_INT16_T, **539**
- C_INT32_T, **539**
- C_INT64_T, **539**
- C_INT8_T, **539**

- C_INT_FAST16_T, [539](#)
- C_INT_FAST32_T, [539](#)
- C_INT_FAST64_T, [539](#)
- C_INT_FAST8_T, [539](#)
- C_INT_LEAST16_T, [539](#)
- C_INT_LEAST32_T, [539](#)
- C_INT_LEAST64_T, [539](#)
- C_INT_LEAST8_T, [539](#)
- C_INTMAX_T, [539](#)
- C_INTPTR_T, [539](#)
- C_LOC, [70](#), [122](#), [372](#), [540](#), [545](#), [608](#)
- C_LONG, [539](#)
- C_LONG_DOUBLE, [539](#)
- C_LONG_DOUBLE_COMPLEX, [539](#)
- C_LONG_LONG, [539](#)
- C_NEW_LINE, [539](#)
- C_NULL_CHAR, [539](#), [546](#)
- C_NULL_FUNPTR, [538](#), [539](#)
- C_NULL_PTR, [538](#), [539](#), [540](#)
- C_PTR, [85](#), [96](#), [115](#), [148](#), [157](#), [158](#), [191](#), [538–540](#), [543](#), [545](#), [548](#), [549](#), [552](#), [588](#), [590](#), [655](#)
- C_PTRDIFF_T, [539](#)
- C_SHORT, [539](#)
- C_SIGNED_CHAR, [539](#)
- C_SIZE_T, [539](#)
- C_SIZEOF, [122](#), [185](#), [546](#)
- C_VERTICAL_TAB, [539](#)
- CALL statement, [228](#), [232](#), [331](#), [344](#), [359](#), [360](#), [369](#), [455](#)
- call-stmt* (R1521), [45](#), [344](#), [345](#), [347](#)
- CASE statement, [220](#)
- case-construct* (R1142), [45](#), [220](#), [221](#)
- case-construct-name*, [220](#), [221](#)
- case-expr* (R1146), [220](#), [221](#), [221](#)
- case-selector* (R1147), [220](#), [221](#)
- case-stmt* (R1144), [220](#), [220](#), [221](#)
- case-value* (R1149), [221](#), [221](#)
- case-value-range* (R1148), [221](#), [221](#)
- CEILING, [407](#)
- CFI_address, [558](#)
- CFI_allocate, [559](#), [566](#)
- CFI_cdesc_t, [551–554](#), [554](#), [555](#), [556](#), [558–560](#), [562–566](#)
- CFI_deallocate, [556](#), [560](#), [566](#)
- CFI_establish, [560](#), [605](#), [664–667](#)
- CFI_is_contiguous, [562](#)
- CFI_section, [562](#), [665](#)
- CFI_select_part, [564](#)
- CFI_setpointer, [565](#), [667](#)
- CHANGE TEAM construct, [55](#), [156](#), [205](#), [207](#), [214](#), [228](#), [357](#), [574](#), [575](#), [591](#)
- CHANGE TEAM statement, [49](#), [55](#), [207](#), [231](#), [241](#), [380](#), [579](#)
- change-team-construct* (R1111), [45](#), [207](#), [207](#)
- change-team-stmt* (R1112), [207](#), [207](#), [208](#)
- changeable mode, [251](#)
- CHAR, [78](#), [407](#)
- char-length* (R723), [76](#), [76](#), [77](#), [85](#), [86](#), [110–112](#), [601](#)
- char-length*, [603](#)
- char-literal-constant* (R724), [60](#), [65](#), [77](#), [275](#), [296](#), [297](#), [592](#)
- char-selector* (R721), [71](#), [76](#), [77](#)
- char-string-edit-desc* (R1322), [295](#), [297](#)
- char-variable* (R905), [146](#), [146](#), [250](#)
- character context, [7](#), [58](#), [62–64](#), [78](#)
- character literal constant, [77](#)
- character sequence type, [24](#), [82](#), [142](#), [144](#), [145](#), [584](#), [587](#)
- character set, [58](#)
- character storage unit, [26](#), [121](#), [142](#), [145](#), [492](#), [582](#), [583](#), [587](#), [589](#)
- character string edit descriptor, [295](#), [314](#)
- character type, [75–79](#)
- CHARACTER_KINDS, [492](#)
- CHARACTER_STORAGE_SIZE, [492](#)
- characteristics, [8](#), [98](#), [195](#), [196](#), [272](#), [273](#), [332](#), [333](#), [336](#), [343](#), [344](#), [354](#), [359](#), [363](#), [365](#), [368](#), [388](#), [394](#), [459](#)
- dummy argument, [332](#)
- procedure, [332](#)
- child data transfer statement, [248](#), [250](#), [262](#), [264](#), [267](#), [268](#), [274](#), [272–276](#), [293](#), [318](#)
- CLASS, [68](#), [68](#), [69](#), [70](#), [272](#)
- CLASS DEFAULT statement, [225](#)
- CLASS IS statement, [225](#), [424](#)
- CLASSOF, [68](#), [69](#)
- CLOSE statement, [244](#), [245](#), [250](#), [252](#), [253](#), [258](#), [258](#), [276](#), [279](#), [626](#)
- close-spec* (R1209), [258](#), [258](#)
- close-stmt* (R1208), [45](#), [258](#), [372](#)
- CMPLX, [190](#), [379](#), [408](#), [501](#)
- CO_BROADCAST, [408](#)
- CO_MAX, [409](#)
- CO_MIN, [409](#)
- CO_REDUCE, [410](#), [596](#)
- CO_SUM, [411](#), [596](#)
- coarray, [8](#), [38](#), [49](#), [54–56](#), [81](#), [86](#), [87](#), [95](#), [115](#), [116](#), [122](#), [124](#), [125](#), [127](#), [128](#), [141](#), [143](#), [149](#)

- 156–159, 163, 164, 188, 189, 191, 194, 195,
205, 207–209, 212, 231–233, 327, 334,
344, 345, 351, 353, 354, 357, 359, 364, 381,
382, 384, 386, 398–402, 413, 435, 441, 454,
455, 485, 486, 489, 490, 495, 550
- established, **8**, 54, 55, 208, 436
- coarray-association* (R1113), 55, 207, **207**, 208
- coarray-name*, 130, 207, 208, 574, 577
- coarray-spec* (R809), 85–87, 110, 111, 115, **115**,
116, 129, 130, 135
- cobound, **8**, 54, 56, 115, 116, 155, 159, 185, 205,
209, 354, 384, 386, 441, 442, 455, 489, 490,
575
- codimension, **8**, 54, 116, 155, 205, 332, 413, 442,
490
- CODIMENSION attribute, 70, 86, 111, 115, **115**, 121,
130
- codimension-decl* (R839), 130, **130**, 205, 207, 208,
574
- codimension-stmt* (R838), 45, **130**, 577
- coindexed object, **8**, 38, 50, 55, 89, 100, 131, 148,
155, 158, 188, 189, 191, 193–195, 204,
230, 233, 237, 344, 345, 349–353, 357, 372,
398–402, 408–411, 422, 454, 455, 486,
497, 540, 542, 544, 545
- coindexed-named-object* (R914), 146, 147, 149, **149**
- collating sequence, **8**, 78, 79, 179, 300, 389, 408,
431, 435, 443, 444, 448–453, 457
- collective subroutine, **27**, 376, 380–382, 408–411,
425, 481, 497, 498, 590
- COMMAND_ARGUMENT_COUNT, 185, 186, **412**, 428
- comment, **63**, **64**, 322
- common association, **144**
- common block, **8**, 41, 46, 52, 111, 113, 114, 125, 126,
130, 131, 141, 143–145, 184, 330, 567, 568,
571–575, 579, 581–584, 588, 601
- common block storage sequence, **144**
- COMMON statement, 141, **143**, 143–145, 206, 328,
330, 573, 582–584, 599
- common-block-name*, 130, 135, 143, 206, 328
- common-block-object* (R878), 143, **143**, 328, 577
- common-stmt* (R877), 45, **143**, 577
- companion processor, **9**, 50, 57, 80, 103, 114, 539,
546, 568, 569
- compatibility
 - FORTTRAN 77, 41
 - Fortran 2003, 39
 - Fortran 2008, 37
 - Fortran 2018, 36
 - Fortran 2023, 36
 - Fortran 90, 40
 - Fortran 95, 40
- COMPILER_OPTIONS, 185, **492**, 608
- COMPILER_VERSION, 185, **493**, 608
- completion step, 50, 258
- complex part designator, **13**, 52, 149
- complex type, 74–75
- complex-literal-constant* (R718), 60, **74**
- complex-part-designator* (R915), 146, 149, **149**, 151,
155
- component, **9**, 54, 80–82, **85**, 100, 136, 573
 - direct, **9**, 79, 80, 89, 351, 500, 548
 - parent, **9**, 91, 95, 97, 100, 584, 615
 - potential subobject, **9**, 38, 79–81, 86, 115, 127,
128, 157, 158, 163, 164, 188, 351, 357, 359,
372, 409, 494–496
 - ultimate, **9**, 38, 39, 79, 80, 117, 120, 122, 141,
143, 161, 186, 188, 212, 272, 345, 349, 371,
410, 583
- component definition statement, 69, **85**
- component keyword, **19**, 56, 91, 100, 573
- component order, **9**, 91, 100, 266
- component specification expression, **25**, 85, 86,
185, **186**
- component-array-spec* (R740), 85, **85**, 86, 87
- component-attr-spec* (R738), 85, **85**, 86, 87, 89
- component-data-source* (R758), 99, **99**, 100, 101
- component-decl* (R739), 77, 85, **85**, 86, 87, 89
- component-def-stmt* (R736), 85, **85**, 86
- component-initialization* (R743), 85, **88**, 89
- component-name*, 85, 89
- component-part* (R735), 80, **85**, 91, 94
- component-spec* (R757), 99, **99**, 100, 186
- computed GO TO statement, 228, **229**, 599, 600
- computed-goto-stmt* (R1160), 46, 229, **229**
- concat-op* (R1012), 60, 168, **168**
- CONCURRENT, **211**
- concurrent-control* (R1126), 199, 201, 211, **211**,
212, 213
- concurrent-header* (R1125), 199–202, 210, 211,
211, 212, 575
- concurrent-limit* (R1127), 172, 201, 211, **211**, 212,
213
- concurrent-locality* (R1129), 211, **211**, 212
- concurrent-step* (R1128), 172, 201, 211, **211**,
212–214
- conditional-arg* (R1526), 345, **345**, 346–348
- conditional-expr* (R1002), 166, 167, **167**, 172

- conformable, **9**, 54, 160, 173, 180, 188, 192, 360, 374, 425, 432, 433, 438, 448, 449, 452, 453, 461, 464, 482, 490, 525
- CONJG, **412**
- connect-spec* (R1205), 253, **253**, 254
- connected, **10**, 244–247, 249, 252, 253, 256–259, 264, 269–272
- connection mode, 251
- consequent* (R1527), 345, **345**, 347
- consequent-arg* (R1528), 345, **345**, 346–348
- constant, **10**, 53, 60, 66
 - integer, 72
 - named, 134
- constant* (R604), 60, **60**, 131, 147, 166
- constant expression, **10**, 30, 40, 41, 66, 67, 76, 84, 87, 89, 108, 112, 116, 119, 121, 131, 132, 134, 141, 184, 186, **186**, 187, **187**, 263, 332–334, 356, 377, 388, 391, 392, 407, 408, 413, 414, 425, 426, 431, 435, 437, 441–443, 446, 448, 452, 458, 467, 470, 472, 475, 478, 481, 489–491, 550
- constant-expr* (R1030), 67, 88, 89, 111, 112, 121, 124, 134, 187, **187**, 221
- constant-subobject* (R851), 131, 132, **132**
- construct
 - ASSOCIATE, 55, **204**, 207, 357, 574, 575, 579, 591
 - BLOCK, 37, 38, 50, 52, 96, 112, 114, 116, 119, 126, 128, 136–140, 162, 184, **206**, 371, 574, 580, 581, 588, 590, 593
 - CHANGE TEAM, 55, 156, 205, **207**, 214, 228, 357, 574, 575, 591
 - CRITICAL, **209**, 214, 228
 - DO, 50, 62, 108, 132, **210**, 228, 267, 598, 617, 618
 - DO CONCURRENT, 37, 38, **210**, 214, 218, 228, 372, 373, 574, 575, 581, 587, 588, 590, 593, 601
 - FORALL, **199**, 372, 373, 574, 575, 587, 599, 601
 - IF, 50, **219**, 598
 - nonblock DO, **599**
 - SELECT CASE, 50, **220**, 600, 617
 - SELECT RANK, 50, 55, 120, 121, 205, **223**, 357, 574, 575, 591
 - SELECT TYPE, 50, 55, 67, 69, 205, **225**, 357, 574, 575, 579, 591
 - WHERE, **197**
- construct association, **5**, 162, 164, 576, **579**, 582, 584, 585
- construct entity, **10**, 128, 139, 204, 206, 207, 212, 216, 225, 571, 572, 574, 581
- construct-name*, 228
- constructor
 - array, 107
 - derived-type, 99
 - structure, 99
- CONTAINS statement, 48, **92**, **369**
- contains-stmt* (R1546), 44, 92, 326, **369**
- contiguous, **10**, 37, 82, 88, **117**, 147, 155, 195, 196, 205, 217, 263, 270, 440, 545, 582, 583, 608
- CONTIGUOUS attribute, 85, 88, 116, **116**, 117, 118, 130, 154, 155, 195, 216, 332, 333, 350–353, 357, 551–553, 583
- CONTIGUOUS statement, **130**
- contiguous-stmt* (R840), 45, **130**
- continuation, 63, 64
- CONTINUE statement, **229**, 598
- continue-stmt* (R1161), 45, 212, **229**
- control character, 58, 77, 243, 247
- control edit descriptor, **295**, 311–314
- control information list, **260**
- control mask, 198
- control-edit-desc* (R1313), 295, **296**
- conversion
 - numeric, 190
- corank, **10**, 54, 56, 87, 115, 116, 118, 148, 155, 157, 166, 205, 209, 332, 346, 348, 352, 413, 435, 436, 441, 442, 454, 455, 485, 490, 575
- corresponding coarrays, **54**, 159
- COS, **412**
- COSD, **412**
- COSH, **413**
- COSHAPE, **413**
- COSPI, **414**
- cosubscript, **11**, 55, 116, 155, 385, 435, 436, 485, 490
- cosubscript* (R927), 148, 155, **155**
- COUNT, 378, **414**
- CPU_TIME, **414**
- CRITICAL construct, **209**, 214, 228
- CRITICAL statement, 181, **209**, 231, 232, 242
- critical-construct* (R1116), 45, 209, **209**
- critical-construct-name*, 209
- critical-stmt* (R1117), 209, **209**, 228
- CSHIFT, **415**
- current record, **247**
- current team, **28**, **49**, 54, 95, 156, 159, 160, 163, 164, 207–209, 233, 234, 236, 238, 241, 380, 381,

408, 424, 425, 430, 435, 436, 455, 460, 481,
484, 485, 490, 493, 498

CURRENT_TEAM, 430, **493**

CYCLE statement, 204, 210, **214**, 215, 601

cycle-stmt (R1135), 46, 214, **214**

D

d (R1310), 296, **296**, 301–306, 309–311, 318

D edit descriptor, 303

data edit descriptor, **295**, 299–311

data entity, **11**, 52, 53, 546

data object, **11**, 46, 47, 49, 52–54, 56

data object designator, **13**, 54, 146

data object reference, **23**, 53, 56

data pointer, **20**, 55, 87, 89, 100, 101, 125, 145, 146,
157, 194, 353, 365, 540, 541, 555, 561, 565,
579, 582, 583, 617, 644

DATA statement, 37, 38, 41, 48, 107, 112, **130**, 145,
206, 330, 459, 574, 575, 577, 586, 599, 600

data transfer, 270

data transfer input statement, **259**

data transfer output statement, **259**

data transfer statement, 41, 61, 243–246, 248–252,
259, 265, 268–270, 275, 278, 279, 281,
290–295, 307, 313, 315, 317, 318, 320, 322,
495, 586, 587, 589, 595, 626, 629, 630

data type, **28**, *see* type

data-component-def-stmt (R737), 85, **85**, 86, 87, 89

data-edit-desc (R1307), 295, **296**

data-i-do-object (R845), 131, **131**, 132

data-i-do-variable (R846), 131, **131**, 132, 187, 574,
575

data-implied-do (R844), 131, **131**, 132, 187, 575

data-pointer-component-name, 193

data-pointer-initialization compatible, **88**

data-pointer-object (R1035), 193, **193**, 194, 200,
591

data-ref (R911), 68, 69, 148, **148**, 149–151, 193,
194, 263, 344, 347, 357, 362

data-stmt (R841), 44, **130**, 335, 371, 577

data-stmt-constant (R849), 107, 131, **131**, 132, 133

data-stmt-object (R843), 130, 131, **131**, 132

data-stmt-repeat (R848), 131, **131**, 132

data-stmt-set (R842), 130, **131**

data-stmt-value (R847), 131, **131**, 132

data-target (R1038), 99–101, 125, 193, **193**, 194,
200, 356, 372

DATE_AND_TIME, **416**

DBLE, 379, **417**

DC edit descriptor, 314

dealloc-opt (R945), 162, **162**, 163, 164

DEALLOCATE statement, **162**, 164, 165, 231, 494,
497, 560, 593

deallocate-stmt (R944), 46, **162**, 591

decimal edit descriptor, 314

decimal edit mode, 255

decimal symbol, **11**, 255, 263, 285, 299, 300,
302–306, 314, 316

decimal-edit-desc (R1318), 296, **297**

DECIMAL= specifier, 254, **255**, 260, 261, **263**, 276,
283, **285**, 314

declaration, **11**, 47, 110–145

declaration-construct (R507), 44, **44**, 206

declaration-type-spec (R703), 68, **68**, 69, 76, 85, 86,
110, 112, 136, 184, 341, 342, 363, 366

declared type, **29**, 69, 70, 88, 100, 101, 108, 110,
148, 150, 158, 161, 167, 180, 182, 188, 191,
192, 194, 195, 205, 224, 226, 227, 278, 339,
344, 345, 347, 348, 352, 362, 370, 424, 450,
455, 471, 494, 495, 575

DEFAULT, **211**, **221**, **225**

default character, **75**

default complex, **74**

default initialization, **11**, 87–90, 99–101, 112, 120,
122, 130, 142, 143, 145, 351, 459, 579, 584,
589

default real, **73**

default-char-constant-expr (R1031), 114, 187, **187**,
260, 261

default-char-expr (R1026), 182, **182**, 187, 229,
253–260, 262–265

default-char-variable (R906), 146, **146**, 156, 254,
283–290

default-initialized, **11**, 89, 123, 364, 580–582,
586–589

DEFERRED attribute, **92**, 93, 97

deferred type parameter, **30**, 38, 67, 69, 77, 101, 125,
145, 150, 157, 158, 161, 164, 165, 188, 189,
195, 217, 232, 317, 332, 333, 352, 364, 388,
442, 459, 481, 540, 542, 551, 579, 585, 602,
603

deferred-coshape-spec (R810), 85, 115, 116, **116**

deferred-shape array, **4**, 120, 127

deferred-shape-spec (R822), 85, 118, 120, **120**, 134

definable, **11**, 123–126, 154, 189, 205, 265, 348,
351, 353, 358, 359, 582, 591

defined, **11**, **12**, 53, 56

defined assignment, **12**, 188, 191, 192, 197, 201,

- 331, 339, 344, 350, 372, 373
- defined assignment statement, 39, **192**, 359, 591
- defined input/output, **12**, 250, 256, 266, 267, **272**, **272**, **272**, **274**, **274**, **275**, **275**, **276**, **276**, 272–278, 290, 311, 318, 319, 323, 324, 331, 337, 339, 344, 359, 372, 495, 647
- defined operation, **12**, 170, **179**, 180, **180**, 181–183, 211, 331, 338, 344, 359, 372
- defined-binary-op* (R1024), 61, 170, **170**, 180, 328
- defined-io-generic-spec* (R1509), 93, 272, 273, 278, 335, **335**, 337, 340
- defined-operator* (R609), **61**, 93, 328, 335, 607
- defined-unary-op* (R1004), 61, 167, **167**, 170, 179, 328
- definition, **12**
- definition of variables, 585
- deleted features, 35, 36, 40–42, 598, 599
- DELIM= specifier, 254, **255**, 260, 261, **263**, 276, 283, **285**, 322, 323, 628
- delimiter mode, 255
- derived type, **29**, 51, 52, 66, 79–102, 108, 548, 549
- derived type definition statement, *see* TYPE statement
- derived type determination, 82
- derived-type type specifier, 69
- derived-type-def* (R726), 44, 70, **80**, 81, 83, 84, 549
- derived-type-spec* (R754), 68–70, 76, 99, **99**, 100, 225, 226, 272, 273, 573
- derived-type-stmt* (R727), 80, **80**, 81, 83, 84, 113, 577
- descendant, **12**, 47, 81, 91, 92, 94, 126, 329, 572
- designator, **12**, 56, 121, 122, 127, 131, 141, 143, 150, **150**, 182, 184, 186, 320, 355, 356, 372, 374
- data object, 146
- designator* (R901), 88, 89, 131, 132, 146, **146**, 148–150, 166, 167, 193, 204, 216, 319, 371, 372, 413, 435, 441, 485, 490
- designator*, 166
- digit*, 33, 58, **58**, 61, 72, 106, 107, 316
- digit-string* (R711), 33, 72, **72**, 73, 301, 302, 308
- digit-string*, 72
- DIGITS, **417**
- DIM, **417**
- DIMENSION attribute, 86, 111, 118, **118**, 126, 133, 144
- DIMENSION statement, **133**, 330
- dimension-stmt* (R852), 45, **133**, 577
- direct access, **245**
- direct access data transfer statement, **264**
- direct component, **9**, 79, 80, 89, 351, 500, 548
- DIRECT= specifier, **283**, **285**
- disassociated, **13**, 55, 70, 88–90, 113, 120, 132, 133, 161, 162, 164, 183, 192, 195, 343, 356, 376, 424, 459, 460, 471, 481, 506, 579, 580, 589
- distinguishable, **340**
- DO CONCURRENT construct, 37, 38, **210**, 214, 218, 228, 372, 373, 574, 575, 581, 587, 588, 590, 593, 601
- DO CONCURRENT statement, 69, 200, **210**
- DO construct, 50, 62, 108, 132, **210**, 228, 267, 598, 617, 618
- DO statement, **210**, 586, 599, 601
- DO WHILE statement, **210**
- do-construct* (R1119), 45, **210**, 212, 214, 228
- do-construct-name*, 211, 212, 214
- do-stmt* (R1120), 210, **211**, 212, 228, 591
- do-variable* (R1124), 108, 131, 211, **211**, 213, 265, 266, 291–293, 317, 586, 589, 591, 626
- DOT_PRODUCT, **418**
- DOUBLE PRECISION, 62, 71, **73**, 80
- DP edit descriptor, 314
- DPROD, **418**
- DSHIFTL, **419**
- DSHIFTR, **419**
- DT edit descriptor, 311
- dtv-type-spec* (R1221), **272**
- dummy argument, **13**, 37, 50, 55, 56, 59, 67, 69–71, 76, 77, 83, 86, 88, 93–96, 98, 99, 111, 114, 115, 118–120, 122–124, 126–128, 131, 133–135, 140, 141, 143, 157, 160, 163, 179, 180, 182, 184, 192, 195, 212, 217, 232, 268, 273, 275, 276, 331–341, 344, 345, 347–356, 364, 368, 369, 371, 372, 374, 375, 459, 494, 551–553, 574, 576, 581, 590, 591, 635, 644
- characteristics of, 332
- restrictions, 357
- dummy data object, **13**, 70, 88, 112, 120–122, 127, 332, 338–340
- assumed-rank, **6**, 54, 70, 94, 95, 116–118, 155, 223, 224, 332, 334, 340, 345, 349–352, 357, 364, 394, 440, 441, 459, 467, 475, 478, 489, 538, 546, 551, 659–661
- dummy function, **13**, 76, 111
- dummy procedure, **21**, 122, 136, 140, 185, 194, 332, 334–336, 340–342, 345, 354, 355, 361, 362, 364, 367, 371, 373, 374, 569, 572, 578
- dummy-arg* (R1539), 366, **366**, 367, 368

dummy-arg-name (R1534), 133, 135, 331, 364, 365, 365, 366, 370, 577
dynamic type, 29, 70, 95, 97, 98, 101, 108, 128, 158, 159, 161, 164, 180, 182, 189, 191, 192, 194, 195, 205, 225, 226, 232, 235, 278, 344, 352, 362, 424, 450, 455, 470, 471, 481, 552, 575, 579, 585, 616, 659

E

e (R1311), 296, 296, 303–306, 309–311, 318

E edit descriptor, 303

edit descriptor, 295

/, 312

;, 312

A, 309

AT, 309

B, 307

BN, 314

BZ, 314

character string, 295, 314

control, 295, 311–314

D, 303

data, 295, 299–311

DC, 314

decimal, 314

DP, 314

DT, 311

E, 303

EN, 304

ES, 305

EX, 306

F, 301

G, 309

H, 598

I, 301

L, 308

LZ, 313

LZP, 313

LZS, 313

O, 307

P, 313

position, 311

RC, 314

RD, 314

RN, 314

round, 314

RP, 314

RU, 314

RZ, 314

S, 313

SP, 313

SS, 313

T, 312

TL, 312

TR, 312

X, 312

Z, 307

effective argument, 13, 67, 70, 77, 117–121, 123, 124, 232, 348–352, 354, 358, 361, 362, 467, 541, 552, 576, 581, 584, 586, 589

effective item, 14, 266, 267, 270, 271, 275, 276, 278, 291, 292, 298, 312, 315–317, 321, 322, 359

effective position, 341

element sequence, 355

ELEMENTAL, 14, 363, 364, 369, 371, 374

elemental, 14, 54, 76, 95, 98, 180, 186, 192, 196–199, 331, 332, 334, 342, 343, 350, 351, 354, 359–361, 369, 374, 376, 382, 403, 404, 456, 506, 508, 510

elemental array assignment (FORALL), 199

elemental assignment, 14, 192

elemental operation, 14, 172, 183, 184, 199

elemental operator, 14, 172, 500

elemental procedure, 14, 54, 183, 194, 342, 345, 356, 361, 363, 373, 374, 374, 376, 377

elemental reference, 14, 199, 351, 359–362, 375

elemental subprogram, 14, 363, 364, 374

ELSE IF statement, 63, 219

ELSE statement, 219

else-if-stmt (R1138), 219, 219

else-stmt (R1139), 219, 219

ELSEWHERE statement, 63, 197

elsewhere-stmt (R1049), 197, 197

EN edit descriptor, 304

ENCODING= specifier, 254, 255, 283, 285, 594

END ASSOCIATE statement, 63, 204

END BLOCK DATA statement, 63, 330

END BLOCK statement, 63, 163, 206

END CRITICAL statement, 63, 181, 209, 231, 232

END DO statement, 63, 212

END ENUM statement, 63, 102

END ENUMERATION TYPE statement, 105

END FORALL statement, 62, 200

END FUNCTION statement, 63, 365

END IF statement, 63, 219, 598

END INTERFACE statement, 63, 334

END MODULE statement, 63, 326

END PROCEDURE statement, 63, 367

- END PROGRAM statement, 63, **325**
- END SELECT statement, 63, **220**, **226**
- END statement, **14**, 49, **49**, 50, 62, 65, 96, 126, 145, 162, 163, 232, 541, 590
- END SUBMODULE statement, 63, **329**
- END SUBROUTINE statement, 62, **366**
- END TEAM statement, 49, 62, **207**, 228, 231, 241, 380
- END TYPE statement, 63, **81**
- END WHERE statement, 63, **197**
- end-associate-stmt* (R1106), 204, **204**, 205, 228
- end-block-data-stmt* (R1422), 44, 49, 330, **330**
- end-block-stmt* (R1110), 206, **206**, 228
- end-change-team-stmt* (R1114), 207, **207**, 208
- end-critical-stmt* (R1118), 209, **209**, 210, 228
- end-do* (R1133), 211, 212, **212**, 215
- end-do-stmt* (R1134), 212, **212**, 228
- end-enum-stmt* (R763), 102, **102**
- end-enumeration-type-stmt* (R770), 104, **105**
- end-forall-stmt* (R1055), 199, 200, **200**
- end-function-stmt* (R1536), 43, 49, 228, 335, 364, 365, **365**, 369
- end-if-stmt* (R1140), 219, **219**, 228
- end-interface-stmt* (R1504), 334, **334**, 335
- end-module-stmt* (R1406), 43, 49, 326, **326**
- end-mp-subprogram-stmt* (R1543), 45, 49, 228, 367, **367**, 369
- end-program-stmt* (R1403), 43, 49, 50, 96, 228, 229, 325, **325**
- end-select-rank-stmt* (R1153), 223, **223**, 224, 228
- end-select-stmt* (R1145), 220, **220**, 221, 228
- end-select-type-stmt* (R1157), 225, 226, **226**, 227, 228
- end-submodule-stmt* (R1419), 44, 49, 329, **329**, 330
- end-subroutine-stmt* (R1540), 43, 49, 228, 335, 366, **366**, 369
- end-type-stmt* (R730), 80, **81**
- end-where-stmt* (R1050), 197, **197**
- END= specifier, 7, 260, 261, 279, 280, **291**
- endfile record, **244**
- ENDFILE statement, 62, 244, 245, 248, 255, 276, 279, **281**, 626
- endfile-stmt* (R1225), 46, **280**, 372
- entity-decl* (R803), 77, 86, 110, 111, **111**, 112, 185, 187, 577
- entity-name*, 130, 134
- ENTRY statement, 48, 141, 179, 180, 192, 326, 331, 336, 363, **367**, 374, 573, 582–584, 599, 601
- entry-name*, 365, 367, 368, 573
- entry-stmt* (R1544), 44, 326, 329, 335, **367**, 368, 573, 577
- enum constructor, 66, 103, 107, 132, 182, 184, 186
- ENUM statement, **102**
- enum type, 66, 67, 69, 70, 85, 102, 145, 174, 221, 300, 301, 308–310, 316, 321, 437, 548
- enum-constructor* (R765), 103, **103**, 132, 166
- enum-def* (R759), 44, 102, **102**, 103, 577
- enum-def-stmt* (R760), 102, **102**
- enum-type-name*, 102, 191, 577
- enum-type-spec* (R764), 68, 69, 102, **102**, 103
- enumeration, **102**
- enumeration constructor, 66, **105**, 132, 182, 184, 186
- enumeration type, 66, 67, 69, 85, **104**, 113, 141, 143, 174, 179, 221, 266, 300, 301, 308, 431, 437, 457, 463
- ENUMERATION TYPE statement, **105**
- enumeration-constructor* (R772), **105**, 132, 166
- enumeration-enumerator-stmt* (R769), 104, **105**
- enumeration-type-def* (R766), 44, **104**
- enumeration-type-name*, 105, 577
- enumeration-type-spec* (R771), 68, 69, 105, **105**
- enumeration-type-stmt* (R767), 104, 105, **105**, 113, 577
- enumerator, **102**
- enumerator* (R762), 102, **102**, 107
- ENUMERATOR statement, **102**
- enumerator-def-stmt* (R761), 102, **102**
- EOR= specifier, 7, 260, 261, 279, 280, **291**, 292, 626
- EOSHIFT, **420**
- EPSILON, **421**
- equiv-op* (R1022), 60, 169, **169**
- equiv-operand* (R1017), 169, **169**
- equivalence association, 142
- EQUIVALENCE statement, 141, **141**, 141–145, 206, 328, 330, 582–584, 599, 601
- equivalence-object* (R876), 141, **141**, 142, 143, 328
- equivalence-set* (R875), 141, **141**, 142
- equivalence-stmt* (R874), 45, **141**, 577
- ERF, **421**
- ERFC, **421**
- ERFC_SCALED, **421**
- ERR= specifier, 7, 254, 258–260, 279, 280, 282–284, **291**
- errmsg-variable* (R931), 156, **156**, 158, 162, 164, 165, 228, 230, 231, 233, 234, 237, 239, 240, 242, 591, 593

- ERRMSG= specifier, 156, 160, 162, **165**, 210, 231, 233, **240**, 587, 593, 605
- error indicator, **558**
- ERROR STOP statement, 50, 51, **229**, 593
- error termination, **50**, 96, 160, 162, 229, 231, 275, 291, 380, 398–403, 422, 423, 455, 457, 464, 469, 592, 595
- error-stop-stmt* (R1163), 46, 96, **229**
- ERROR_UNIT, 250–252, 256, **493**
- ES edit descriptor, 305
- established coarray, **8**, 54, 55, 156, 208, 436
- evaluation
- operations, 171
 - optional, 181
 - parentheses, 181
- EVENT POST statement, 231, 237, **237**, 238, 241, 422, 493, 494, 588, 591, 594
- event variable, **31**, 50, 232, 237, 238, 241, 422, 493, 494, 588
- EVENT WAIT statement, 231, 237, **237**, 241, 493, 494, 588, 591, 594
- event-post-stmt* (R1174), 46, **237**
- event-variable* (R1175), 237, **237**, 240, 241, 494, 591
- event-wait-spec* (R1177), 230, 237, **237**
- event-wait-stmt* (R1176), 46, 237, **237**
- EVENT_QUERY, **422**, 606, 623, 653
- EVENT_TYPE, 81, 123, 158, 237, **493**
- EX edit descriptor, 306
- executable construct, 203
- executable statement, **25**, 47
- executable-construct* (R514), 44, **45**, 368
- EXECUTE_COMMAND_LINE, 386, **422**
- execution control, 203
- execution-part* (R509), 43, **44**, 45, 325, 364–367
- execution-part-construct* (R510), 44, **44**, 203
- exist, 245, 252
- EXIST= specifier, 283, **286**
- EXIT statement, 204, 215, **228**
- exit-stmt* (R1158), 46, 228, **228**
- EXP, **423**
- explicit formatting, 294–314
- explicit initialization, **14**, 89, 90, 111–113, 130, 579, 584, 586
- explicit interface, **17**, 39, 88, 92, 136, 140, 195, 196, 332–335, 337, 342, 343, 345, 347, 354, 355, 370, 371, 572, 574, 591, 635
- explicit-bounds-expr* (R819), 118, **118**, 119
- explicit-coshape-spec* (R811), 115, 116, **116**
- explicit-shape array, **4**, 70, 87, 115, 118, 119, 187, 350, 352, 355, 550, 552, 553
- explicit-shape-bounds-spec* (R818), 118, **118**, 119
- explicit-shape-spec* (R815), 85, 86, 113, 118, **118**, 119–121, 143
- EXPONENT, **423**, 514
- exponent* (R717), 73, **74**
- exponent-letter* (R716), 73, 74, **74**
- expr* (R1023), 34, 95, 99, 100, 103, 107, 157, 166, 167, 170, **170**, 172, 182, 184, 186–195, 198–200, 204, 207, 217, 221, 265, 344, 345, 347, 370, 372, 536, 590
- expression, **166**, 166–187
- component specification, **25**, 85, 86, **185**, **186**
 - constant, **10**, 30, 40, 41, 66, 67, 76, 84, 87, 89, 108, 112, 116, 119, 121, 131, 132, 134, 141, 184, 186, **186**, 187, **187**, 263, 332–334, 356, 377, 388, 391, 392, 407, 408, 413, 414, 425, 426, 431, 435, 437, 441–443, 446, 448, 452, 458, 467, 470, 472, 475, 478, 481, 489–491, 550
 - specification, **25**, 49, 50, 84, 96, 114, 150, **184**, **185**, **185**, 186, 206, 368, 501, 600
- extended real model, 379
- extended type, **29**, 84, 91, 95–97, 584, 609, 613
- extended-intrinsic-op* (R610), 61, **61**
- EXTENDS attribute, 96, **96**, 549
- EXTENDS_TYPE_OF, 185, **424**
- extensible type, **29**, 68, 80, 81, 88, 96, 272, 424, 471, 615, 649
- extension operation, 170
- extension type, **29**, 70, 96–98, 226, 352, 424, 649
- extent, **15**, 54, 350
- EXTERNAL attribute, 35, 36, 122, **122**, 125, 134, 136, 138, 194, 326, 330, 336, 341, 355, 360, 361, 576–578, 642
- external file, **15**, 40, 243–247, 249–253, 258, 262, 263, 282, 299, 311, 322, 373, 569, 626, 654
- external input/output unit, **15**, 571
- external linkage, 114, **538**, 567–569
- external procedure, **21**, 35, 36, 47, 92, 122, 136, 194, 236, 331, 332, 334, 336, 337, 341, 342, 345, 354, 355, 362, 571, 572, 576, 578, 635, 639, 642
- EXTERNAL statement, 122, **341**
- external subprogram, **27**, 46, 331
- external unit, **15**, 250–253, 269, 275, 276, 287, 293, 493, 494, 497
- external-name*, 341

external-stmt (R1511), 45, **341**

external-subprogram (R503), 43, **43**, 138, 368

F

F edit descriptor, 301

F_C_STRING, **546**

FAIL IMAGE statement, **230**

fail-image-stmt (R1165), 46, **230**

failed image, **16**, 50, 156, 159, 163, 164, 208, 239, 381, 455

FAILED_IMAGES, **424**

field, **297**

file

connected, 252

external, **15**, 40, 243–247, 249–253, 258, 262, 263, 282, 299, 311, 322, 373, 569, 626, 654

internal, **18**, 243, 249–251, 253, 262, 267, 269, 270, 272, 274, 276, 290, 292, 311, 312, 586, 589

file access method, 244–247

file connection, 250–259

file inquiry statement, **282**

file position, 244, 247

file positioning statement, 244, **280**

file storage unit, **15**, 243, 244, 246–249, 257, 263, 264, 270, 271, 281, 288–290, 494, 582

file-name-expr (R1206), 254, **254**, 256, 283, 284, 286

file-unit-number (R1202), 250, **250**, 253, 254, 258, 261, 274, 279, 280, 282–290, 372, 495

FILE= specifier, 253, 254, 256, **256**, 258, 283, 284, **284**, 591, 627

FILE_STORAGE_SIZE, **494**

FINAL statement, **94**

final subroutine, **15**, 39, 93–96, 154, 349, 613

final-procedure-stmt (R753), 92, **94**

final-subroutine-name, 94

finalizable, **15**, 39, 95, 120, 122, 163, 212

finalization, **15**, 95, 96, 154, 200, 331, 344, 359, 371, 372

FINDLOC, **425**

fixed source form, 64, **64**

FLOOR, **426**

FLUSH statement, 245, 279, **282**, 292

flush-spec (R1229), 282, **282**

flush-stmt (R1228), 46, **282**, 372

FMT= specifier, 260, **262**

FORALL construct, **199**, 372, 373, 574, 575, 587, 599, 601

FORALL statement, 69, 172, **201**, 574, 575, 586

forall-assignment-stmt (R1054), 172, 199, 200, **200**, 201, 373

forall-body-construct (R1053), 199, **199**, 200, 201

forall-construct (R1051), 45, 199, **199**, 200, 201

forall-construct-name, 199, 200

forall-construct-stmt (R1052), 199, **199**, 200, 228

forall-stmt (R1056), 46, 199, 201, **201**, 228

FORM TEAM statement, 49, 208, 231, **238**, 241, 588, 590, 591, 594

form-team-spec (R1182), 238, **238**

form-team-stmt (R1179), 46, **238**

FORM= specifier, 254, **256**, 283, **286**

format (R1215), 259–262, **262**, 269, 294, 295

format control, **297**

format descriptor, *see* edit descriptor

FORMAT statement, 35, 48, 61, 206, 262, 294, **294**, 326

format-item (R1304), 295, **295**

format-items (R1303), 294, 295, **295**

format-specification (R1302), 294, **294**

format-stmt (R1301), 44, 294, **294**, 326, 329, 335

FORMATTED, 272, 273, **335**

formatted data transfer, 271

formatted input/output statement, 243, **261**

formatted record, **243**

FORMATTED= specifier, 283, **286**

formatting

explicit, 294–314

list-directed, 272, 315–319

namelist, 272, 319–324

forms, 244

Fortran 2003 compatibility, 39

Fortran 2008 compatibility, 37

Fortran 2018 compatibility, 36

Fortran 2023 compatibility, 36

FORTTRAN 77 compatibility, 41

Fortran 90 compatibility, 40

Fortran 95 compatibility, 40

Fortran character set, **58**, 75

FRACTION, **427**

free source form, 62, **62**

function, **15**

intrinsic, 376

intrinsic elemental, 376

intrinsic inquiry, 376

function reference, **23**, 52, 53, 359

function result, **15**, 39, 77, 110, 136, 141, 143, 162, 332, 365, 368, 374, 551, 573, 584, 589

- FUNCTION statement, 69, 70, 136, 179, 180, 184, 325, 363, **364**, 367–369, 573
- function-name*, 111, 335, 336, 364, 365, 368, 370, 573, 577
- function-reference* (R1520), 100, 111, 146, 166, 167, **344**, 347, 359
- function-stmt* (R1533), 43, 334–336, 363, 364, **364**, 365, 573, 577
- function-subprogram* (R1532), 27, 43–45, 326, **364**, 367
- ## G
- G edit descriptor, 309
- GAMMA, **427**
- generic identifier, **16**, 327, 335, 337, 338, 340, 361, 376, 571, 576
- generic interface, **17**, 93, 94, 97, 102, 124, 179, 180, 192, 278, 327, 328, 337, **337**, 338, 339, 360, 361, 572, 650
- generic interface block, **18**, 336, 337, 340
- generic procedure reference, 340
- GENERIC statement, **92**, **93**, 337, **337**, 340, 360
- generic-name*, 92, 93, 335, 574, 577
- generic-spec* (R1508), 92–94, 97, 128, 180, 192, 327, 328, 334, 335, **335**, 337, 360, 574, 577
- generic-stmt* (R1510), 44, **337**
- GET_COMMAND, **427**
- GET_COMMAND_ARGUMENT, **428**
- GET_ENVIRONMENT_VARIABLE, **429**, 596
- GET_TEAM, 185, 186, **430**, 493, 494, 497
- global entity, **571**
- global identifier, **571**
- GO TO statement, 63, 228, **228**
- goto-stmt* (R1159), 46, 228, **228**
- graphic character, 58, 77, 322
- ## H
- halting mode, 499, 504, **504**, 508, 510, 525, 531, 532, 569, 597
- hex-constant* (R776), 106, **107**
- hex-digit* (R777), 107, **107**, 308
- hex-digit-string* (R1323), 308, **308**
- host, **16**, 47, 329, 370, 574, 576–578
- host association, **5**, 21, 47, 69, 70, 77, 114, 128, 131, 136, 144, 184, 185, 194, 329, 331, 356, 370, 372–374, 574, 576–579, 581, 582, 584, 585, 669
- host instance, **16**, 195, 346, 355, 367, 394, 576, 581, 584, 590
- host scoping unit, **16**, 47, 136, 138, 139, 360, 361, 578, 584
- HUGE, **431**
- HYPOT, **431**
- ## I
- I edit descriptor, 301
- IACHAR, 79, 191, **431**
- IALL, **432**
- IAND, 211, 216, 398, **432**
- IANY, **433**
- IBCLR, **433**
- IBITS, **434**
- IBSET, **434**
- ICHAR, 78, **434**
- id-variable* (R1214), 260, **260**
- ID= specifier, 260, 261, **263**, **279**, 283, 284, **286**, 591, 629, 630
- IEEE infinity, **16**
- IEEE NaN, **16**, 179, 502, 527
- IEEE_ALL, **500**
- IEEE_ARITHMETIC, 37, 38, 40, 184–186, 215, 388, 499–536
- IEEE_AWAY, **503**, 511
- IEEE_CLASS, 506, 508, **508**
- IEEE_CLASS_TYPE, **500**, 509, 536
- IEEE_COPY_SIGN, 505, 506, **509**
- IEEE_DATATYPE, **500**
- IEEE_DENORMAL, **500**
- IEEE_DIVIDE, **500**
- IEEE_DIVIDE_BY_ZERO, **500**
- IEEE_DOWN, **500**, **503**
- IEEE_EXCEPTIONS, 184–186, 215, 499–536
- IEEE_FEATURES, 499–500
- IEEE_FEATURES_TYPE, **500**
- IEEE_FLAG_TYPE, **500**, 510, 525, 531
- IEEE_FMA, 506, **509**
- IEEE_GET_FLAG, 215, 502, 508, **510**, 536, 537, 597
- IEEE_GET_HALTING_MODE, 215, 508, 510, **510**
- IEEE_GET_MODES, 504, 508, **510**, 511, 525, 526
- IEEE_GET_ROUNDING_MODE, 503, 506, 511, **511**, 526
- IEEE_GET_STATUS, 215, 502, 508, **511**, 512, 526, 537, 597
- IEEE_GET_UNDERFLOW_MODE, 506, **512**, 527
- IEEE_HALTING, **500**
- IEEE_INEXACT, **500**
- IEEE_INEXACT_FLAG, **500**
- IEEE_INF, **500**

IEEE_INT, 506, 512
 IEEE_INVALID, 500
 IEEE_INVALID_FLAG, 500
 IEEE_IS_FINITE, 507, 513
 IEEE_IS_NAN, 507, 513
 IEEE_IS_NEGATIVE, 507, 513
 IEEE_IS_NORMAL, 507, 514
 IEEE_LOGB, 505, 507, 514
 IEEE_MAX, 507, 514
 IEEE_MAX_MAG, 507, 515
 IEEE_MAX_NUM, 37, 507, 515
 IEEE_MAX_NUM_MAG, 37, 507, 516
 IEEE_MIN, 507, 516
 IEEE_MIN_MAG, 507, 517
 IEEE_MIN_NUM, 37, 507, 517
 IEEE_MIN_NUM_MAG, 37, 507, 518
 IEEE_MODES_TYPE, 500, 504, 511, 525
 IEEE_NAN, 500
 IEEE_NEAREST, 500, 503
 IEEE_NEGATIVE_DENORMAL, 500
 IEEE_NEGATIVE_INF, 500
 IEEE_NEGATIVE_NORMAL, 500
 IEEE_NEGATIVE_SUBNORMAL, 500, 500, 509, 513
 IEEE_NEGATIVE_ZERO, 500
 IEEE_NEXT_AFTER, 507, 518
 IEEE_NEXT_DOWN, 507, 519, 519
 IEEE_NEXT_UP, 507, 519
 IEEE_OTHER, 500, 503
 IEEE_OTHER_VALUE, 500
 IEEE_OVERFLOW, 500
 IEEE_POSITIVE_DENORMAL, 500
 IEEE_POSITIVE_INF, 500
 IEEE_POSITIVE_NORMAL, 500
 IEEE_POSITIVE_SUBNORMAL, 500, 500, 509, 513
 IEEE_POSITIVE_ZERO, 500
 IEEE_QUIET_EQ, 507, 520
 IEEE_QUIET_GE, 507, 520
 IEEE_QUIET_GT, 507, 520
 IEEE_QUIET_LE, 507, 521
 IEEE_QUIET_LT, 507, 521
 IEEE_QUIET_NAN, 500
 IEEE_QUIET_NE, 507, 522
 IEEE_REAL, 507, 522
 IEEE_REM, 505, 507, 522
 IEEE_RINT, 505, 507, 523
 IEEE_ROUND_TYPE, 500, 511, 512, 523, 526, 533
 IEEE_ROUNDING, 500
 IEEE_SCALB, 37, 38, 40, 505, 507, 523
 IEEE_SELECTED_REAL_KIND, 507, 524
 IEEE_SET_FLAG, 502, 508, 512, 525, 526, 537, 597
 IEEE_SET_HALTING_MODE, 215, 502, 508, 510, 525, 532, 537, 597
 IEEE_SET_MODES, 215, 504, 508, 511, 525, 526
 IEEE_SET_ROUNDING_MODE, 215, 503, 507, 511, 526, 526
 IEEE_SET_STATUS, 215, 502, 503, 508, 512, 526, 526, 537, 597
 IEEE_SET_UNDERFLOW_MODE, 215, 507, 511, 512, 526, 527
 IEEE_SIGNALING_EQ, 507, 527
 IEEE_SIGNALING_GE, 507, 527
 IEEE_SIGNALING_GT, 507, 528
 IEEE_SIGNALING_LE, 507, 528
 IEEE_SIGNALING_LT, 507, 529
 IEEE_SIGNALING_NAN, 500
 IEEE_SIGNALING_NE, 507, 529
 IEEE_SIGNBIT, 507, 529
 IEEE_SQRT, 500
 IEEE_STATUS_TYPE, 500, 504, 511, 526, 537
 IEEE_SUBNORMAL, 500
 IEEE_SUPPORT_DATATYPE, 499, 501, 507–509, 512, 514–523, 526–530, 530, 534, 536
 IEEE_SUPPORT_DENORMAL, 507, 530
 IEEE_SUPPORT_DIVIDE, 507, 531, 534
 IEEE_SUPPORT_FLAG, 508, 531, 534
 IEEE_SUPPORT_HALTING, 508, 531, 534
 IEEE_SUPPORT_INF, 505, 507, 519, 532, 534, 536
 IEEE_SUPPORT_IO, 507, 532
 IEEE_SUPPORT_NAN, 502, 505, 507, 532, 534, 536
 IEEE_SUPPORT_ROUNDING, 507, 526, 533, 534
 IEEE_SUPPORT_SQRT, 507, 533, 534
 IEEE_SUPPORT_STANDARD, 506, 507, 534
 IEEE_SUPPORT_SUBNORMAL, 505–507, 509, 519, 530, 534, 534, 536
 IEEE_SUPPORT_UNDERFLOW_CONTROL, 507, 535
 IEEE_TO_ZERO, 500, 503
 IEEE_UNDERFLOW, 500
 IEEE_UNDERFLOW_FLAG, 500
 IEEE_UNORDERED, 505, 507, 535
 IEEE_UP, 500, 503
 IEEE_USUAL, 500
 IEEE_VALUE, 507, 535
 IEOR, 211, 216, 402, 435
 IF construct, 50, 219, 598
 IF statement, 171, 220
 if-construct (R1136), 45, 219, 219
 if-construct-name, 219
 if-stmt (R1141), 46, 220, 220

if-then-stmt (R1137), 219, **219**, 228
imag-part (R720), 74, **74**
 image, **1**, **16**, 38, 49–51, 54, 55, 95, 115, 156, 158–160, 163, 191, 194, 195, 208–210, 229–236, 238–242, 244, 245, 250–252, 287, 344, 349, 354, 357, 359, 376, 380, 381, 385, 386, 398–403, 415–417, 429, 436, 455, 460, 465, 466, 482, 483, 485, 490, 494, 498, 504, 571, 579, 581, 588, 590
 active, **16**, 164
 failed, **16**, 159, 163, 164, 239, 455
 stopped, **16**, 159, 163, 164, 455
 image control statement, **17**, 49, 181, 209, 215, 230, **231**, 232, 233, 236, 240–242, 359, 372, 380, 381, 425, 481, 497
 image index, **16**, 49, 55, 155, 156, 234, 238, 244, 354, 385, 435, 436, 465, 485, 490, 571
image-selector (R926), 49, 148, 149, 155, **155**, 156, 319
image-selector-spec (R928), 155, **155**
image-set (R1171), 234, **234**
 IMAGE_INDEX, **435**
 IMAGE_STATUS, **436**
 imaginary part, **74**
 implicit interface, **17**, 86, 196, 326, 342–345, 354, 544, 578
 IMPLICIT NONE statement, **136**
 IMPLICIT statement, **136**, 141, 330
implicit-none-spec (R870), 136, **136**
implicit-part (R505), 44, **44**
implicit-part-stmt (R506), 44, **44**
implicit-spec (R868), 136, **136**
implicit-stmt (R867), 44, 136, **136**
 implied-shape array, **121**
implied-shape-or-assumed-size-spec (R826), **118**, 120, **120**, 121
implied-shape-spec (R827), 118, 121, **121**
 IMPORT statement, 48, **138**, 206, 571, 574, 578
 import-name, **138**, 139
import-stmt (R871), 44, **138**, 206
 IMPURE, **363**, 364, 369, 371, 374
 IN, **122**
 INCLUDE line, 62, **65**
 inclusive scope, **17**, 207, 228, 229, 254, 259, 260, 262, 279, 280, 282, 284, 345, 369, 571, 572
 INDEX, **436**
index-name, 200–202, 210–214, 574, 575, 587
 inherit, **17**, 80, 92, 93, 95, 97, 98, 584, 615
 inheritance association, **5**, 57, 97, 99, 582, 584
 initial team, **28**, 49, **49**, 55, 156, 251, 386, 430, 436, 460, 465, 484, 494
initial-data-target (R744), 38, 88, **88**, 89, 111, 113, 125, 131–133
initial-proc-target (R1518), 89, 342, **342**, 343
 INITIAL_TEAM, 430, **494**
 initialization, **112**
 default, **11**, 87–90, 99–101, 112, 120, 122, 130, 142, 143, 145, 351, 579, 584, 589
 explicit, **14**, 89, 90, 111–113, 130, 579, 584, 586
initialization (R805), 107, 111, **111**, 112, 187
 INOUT, 63, **122**
 input statement, **259**, 260, 626
input-item (R1216), 259, 260, 265, **265**, 266, 278, 293, 591
 input/output editing, 294–324
 input/output list, 265
 input/output statement, 586
 input/output statements, 243–293
 input/output unit, **31**, 49
 INPUT_UNIT, 250–252, 256, 274, **494**
 INQUIRE statement, 40, 245, 247, 249, 250, 252, 264, 275, 276, 279, **282**, 292, 293, 495, 587, 589, 591, 595, 625
inquire-spec (R1231), 283, **283**, 284, 293
inquire-stmt (R1230), 46, **283**, 372
 inquiry function, **17**, 116, 120, 122, 148, 160, 185, 348, 349, 376–379, 382, 392, 394, 406, 413, 417, 421, 424, 431, 439–442, 448, 451, 457, 463, 464, 467, 470, 475, 478, 481, 486, 489, 504–506, 530–535, 546
 inquiry, type parameter, 150
 instance, 366
 INT, **133**, 190, 378, 419, 432, 435, **437**, 438, 451
int-constant (R607), 60, **60**, 131
int-constant-expr (R1032), 71, 76, 83, 84, 102, 103, 126, 127, 131, 187, **187**, 223, 224
int-constant-name, 72
int-constant-subobject (R850), 131, 132, **132**
int-expr (R1027), 49, 67, 105, 108, 118, 147, 151, 152, 155–157, 172, 182, **182**, 184, 186, 187, 211, 213, 229, 230, 234, 237, 238, 250, 254, 260, 265, 268, 279, 283, 369
int-literal-constant (R708), 60, 72, **72**, 76, 295–297
int-variable (R907), 146, **146**, 164, 254, 260, 261, 283, 284, 287–292, 595
int-variable-name, 211
 INT16, **494**
 INT32, **494**

- INT64, [494](#)
- INT8, [494](#)
- integer constant, [72](#)
- integer editing, [301](#)
- integer model, [379](#)
- integer type, [71–72](#)
- integer-type-spec* (R705), [69](#), [71](#), [71](#), [83](#), [108](#), [131](#), [211](#), [575](#)
- INTEGER_KINDS, [495](#)
- INTENT (IN) attribute, [122](#), [122](#), [123](#), [124](#), [128](#), [212](#), [338–340](#), [349](#), [352](#), [353](#), [355](#), [357](#), [371–373](#), [377](#), [398–403](#), [409–411](#), [422](#), [423](#), [428](#), [429](#), [456](#), [465](#), [466](#), [508](#), [540–543](#), [566](#), [590](#), [644](#), [657](#)
- INTENT (INOUT) attribute, [37](#), [122](#), [123](#), [124](#), [127](#), [217](#), [339](#), [345](#), [351](#), [353](#), [360](#), [373–375](#), [388](#), [398–402](#), [408–411](#), [423](#), [428](#), [429](#), [454–456](#), [494](#), [496](#), [497](#), [591](#), [656](#)
- INTENT (OUT) attribute, [37–39](#), [70](#), [94](#), [96](#), [120](#), [122](#), [122](#), [123](#), [124](#), [127](#), [163](#), [184](#), [339](#), [345](#), [351](#), [353](#), [360](#), [371](#), [373–375](#), [388](#), [398–403](#), [409–411](#), [415](#), [416](#), [422](#), [423](#), [427–429](#), [455–457](#), [463](#), [466](#), [482](#), [483](#), [510–512](#), [540](#), [542](#), [543](#), [566](#), [580](#), [582](#), [586](#), [587](#), [589](#), [591](#), [657](#)
- INTENT attribute, [122](#), [122](#), [123](#), [124](#), [133](#), [216](#)
- INTENT statement, [133](#), [206](#)
- intent-spec* (R829), [110](#), [122](#), [133](#), [342](#)
- intent-stmt* (R853), [45](#), [133](#)
- interface, [17](#), [47](#), [53](#), [56](#), [86](#), [93](#), [124](#), [272–274](#), [311](#), [332](#), [333](#), [344](#), [345](#), [354](#), [355](#), [359](#), [361](#), [367](#), [369–371](#), [551–553](#), [568](#), [569](#), [635](#)
 - abstract, [17](#), [327](#), [333](#), [336](#), [342](#), [364](#), [572](#), [576](#)
 - explicit, [17](#), [39](#), [88](#), [92](#), [136](#), [140](#), [195](#), [196](#), [332–335](#), [337](#), [342](#), [343](#), [345](#), [347](#), [354](#), [355](#), [370](#), [371](#), [572](#), [574](#), [591](#), [635](#)
 - generic, [17](#), [93](#), [94](#), [97](#), [102](#), [124](#), [179](#), [180](#), [192](#), [278](#), [327](#), [328](#), [337](#), [337](#), [338](#), [339](#), [360](#), [361](#), [572](#)
 - implicit, [17](#), [86](#), [196](#), [326](#), [342–345](#), [354](#), [544](#), [578](#)
 - procedure, [333](#)
 - specific, [18](#), [278](#), [335](#), [336](#), [337](#), [342](#), [360](#), [361](#)
- interface block, [18](#), [47](#), [272](#), [278](#), [327](#), [334–337](#), [360](#), [361](#), [635](#)
- interface body, [18](#), [24](#), [48](#), [116](#), [119](#), [122](#), [136](#), [184](#), [334](#), [334](#), [363](#), [364](#), [367](#), [369](#), [553](#), [574](#), [576](#), [577](#), [635](#)
- INTERFACE statement, [334](#), [635](#)
- interface-block* (R1501), [44](#), [334](#), [335](#)
- interface-body* (R1505), [334](#), [334](#), [335](#)
- interface-name* (R1516), [92](#), [93](#), [341](#), [342](#), [342](#)
- interface-specification* (R1502), [334](#), [334](#), [335](#)
- interface-stmt* (R1503), [334](#), [334](#), [335](#), [337](#), [577](#)
- internal file, [18](#), [243](#), [249–251](#), [253](#), [262](#), [267](#), [269](#), [270](#), [272](#), [274](#), [276](#), [290](#), [292](#), [311](#), [312](#), [586](#), [589](#)
- internal procedure, [21](#), [47](#), [194](#), [331–334](#), [345](#), [346](#), [354](#), [362](#), [364](#), [367](#), [394](#), [569](#), [572](#), [574](#), [577](#)
- internal subprogram, [27](#), [47–49](#), [136](#), [139](#), [331](#), [360](#), [576](#)
- internal unit, [19](#), [250](#), [253](#), [269](#), [275](#), [284](#), [293](#), [495](#)
- internal-file-variable* (R1203), [250](#), [250](#), [261](#), [293](#), [591](#)
- internal-subprogram* (R512), [44](#), [44](#)
- internal-subprogram-part* (R511), [43](#), [44](#), [45](#), [325](#), [364–367](#)
- interoperable, [18](#), [102](#), [103](#), [114](#), [364](#), [370](#), [538](#), [540–542](#), [545–553](#), [567](#), [568](#)
- interoperable enumeration, [102](#), [538](#)
- interoperate, [538](#)
- intrinsic, [18](#), [51](#), [52](#), [54](#), [55](#), [57](#), [67](#), [70](#), [95](#), [107](#), [122](#), [332](#), [333](#), [353](#), [361](#), [362](#), [374](#), [492](#), [572](#), [574](#)
- intrinsic assignment statement, [39](#), [100](#), [155](#), [163](#), [165](#), [182](#), [188](#), [193](#), [194](#), [216](#), [242](#), [250](#), [282](#), [293](#), [322](#), [372](#), [377](#), [381](#), [388](#), [408](#), [486](#), [586](#), [593](#)
- INTRINSIC attribute, [122](#), [124](#), [124](#), [125](#), [326](#), [343](#), [360](#), [361](#), [578](#)
- intrinsic function, [376](#)
- INTRINSIC module nature, [327](#)
- intrinsic operation, [171](#), [173–179](#)
- intrinsic procedure, [376–491](#)
- INTRINSIC statement, [330](#), [343](#)
- intrinsic subroutines, [376](#)
- intrinsic type, [29](#), [51](#), [66](#), [71–79](#), [552](#), [556](#)
- intrinsic-operator* (R608), [60](#), [61](#), [167](#), [170](#), [173](#), [179](#), [180](#), [338](#)
- intrinsic-procedure-name*, [343](#), [577](#)
- intrinsic-stmt* (R1519), [45](#), [343](#), [577](#)
- intrinsic-type-spec* (R704), [68](#), [69](#), [71](#), [77](#)
- io-control-spec* (R1213), [251](#), [259](#), [260](#), [260](#), [261](#), [264](#), [274](#), [293](#)
- io-implied-do* (R1218), [265](#), [265](#), [266](#), [267](#), [270](#), [293](#), [586](#), [589](#), [591](#), [626](#)
- io-implied-do-control* (R1220), [265](#), [265](#), [268](#)
- io-implied-do-object* (R1219), [265](#), [265](#), [270](#)
- io-unit* (R1201), [250](#), [250](#), [251](#), [260](#), [261](#), [372](#)

- IOLENGTH= specifier, [249](#), [282](#), [290](#)
iomsg-variable (R1207), [254](#), [254](#), [258](#), [260](#), [279](#), [280](#), [282](#), [283](#), [291–293](#), [587](#)
 IOMSG= specifier, [254](#), [258](#), [260](#), [279](#), [280](#), [282](#), [283](#), [291](#), [292](#), [293](#), [587](#)
 IOR, [211](#), [216](#), [402](#), [438](#)
 IOSTAT= specifier, [254](#), [258](#), [260](#), [275](#), [279](#), [280](#), [282](#), [283](#), [291](#), [292](#), [292](#), [440](#), [495](#), [586](#), [595](#), [626](#)
 IOSTAT_END, [275](#), [292](#), [495](#)
 IOSTAT_EOR, [275](#), [292](#), [495](#)
 IOSTAT_INQUIRE_INTERNAL_UNIT, [275](#), [292](#), [495](#), [498](#)
 IPARITY, [438](#)
 IS_CONTIGUOUS, [70](#), [439](#)
 IS_IOSTAT_END, [440](#)
 IS_IOSTAT_EOR, [440](#)
 ISHFT, [439](#)
 ISHFTC, [439](#)
 ISO 10646 character, [19](#), [75](#), [79](#), [188](#), [249](#), [250](#), [255](#), [267](#), [300](#), [315](#), [316](#), [457](#), [472](#)
 ISO_C_BINDING, [70](#), [85](#), [96](#), [103](#), [115](#), [122](#), [148](#), [157](#), [158](#), [184](#), [185](#), [191](#), [372](#), [492](#), [538–548](#), [588](#), [590](#), [608](#)
 ISO_Fortran_binding.h, [553](#)
 ISO_FORTRAN_ENV, [37](#), [81](#), [85](#), [115](#), [123](#), [148](#), [156–158](#), [164](#), [185](#), [191](#), [208](#), [229–231](#), [237–239](#), [241](#), [249–251](#), [256](#), [269](#), [274](#), [275](#), [292](#), [380](#), [381](#), [398–402](#), [424](#), [425](#), [430](#), [436](#), [455](#), [460](#), [481](#), [482](#), [484](#), [485](#), [492–498](#), [588](#), [590](#), [596](#), [626](#)
- ## K
- k* (R1314), [296](#), [296](#), [304](#), [310](#), [313](#)
 keyword, [19](#)
 argument, [19](#), [56](#), [333](#), [337](#), [347](#), [376](#), [381](#), [506](#), [572](#), [574](#), [635](#)
 component, [19](#), [56](#), [91](#), [100](#), [573](#)
 statement, [19](#), [56](#)
 type parameter, [19](#), [56](#), [99](#)
keyword (R516), [56](#), [56](#), [99](#), [100](#), [344](#), [345](#)
 KIND, [72–75](#), [79](#), [84](#), [104](#), [150](#), [190](#), [191](#), [440](#)
 kind type parameter, [30](#), [35](#), [51](#), [67](#), [71–75](#), [77](#), [79](#), [84](#), [94](#), [95](#), [103](#), [104](#), [108](#), [167](#), [186–190](#), [319](#), [339](#), [345](#), [347–349](#), [364](#), [418](#), [424](#), [471](#), [494](#), [496](#), [497](#), [538](#), [539](#), [547](#), [608](#)
kind-param (R709), [72](#), [72](#), [73](#), [74](#), [76](#), [77](#), [79](#)
kind-selector (R706), [34](#), [71](#), [71](#), [72](#), [79](#)
- ## L
- L edit descriptor, [308](#)
 label, *see* statement label
label (R611), [61](#), [61](#), [211](#), [212](#), [228](#), [229](#), [254](#), [258–260](#), [262](#), [279](#), [280](#), [282–284](#), [291](#), [292](#), [345](#)
label-do-stmt (R1121), [211](#), [211](#), [212](#)
language-binding-spec (R808), [110](#), [111](#), [114](#), [130](#), [364](#)
 LBOUND, [70](#), [189](#), [195](#), [205](#), [224](#), [440](#)
lbracket (R780), [85](#), [107](#), [107](#), [110](#), [111](#), [129](#), [130](#), [135](#), [155](#), [157](#)
 LCOBOUND, [441](#)
 leading zero mode, [256](#), [264](#), [313](#)
leading-zero-edit-desc (R1319), [296](#), [297](#)
 LEADING_ZERO= specifier, [254](#), [256](#), [260](#), [261](#), [264](#), [283](#), [286](#), [313](#)
 LEADZ, [442](#)
 left tab limit, [312](#)
 LEN, [150](#), [442](#)
 LEN_TRIM, [442](#)
 length type parameter, [30](#), [51](#), [67](#), [79](#), [88](#), [108](#), [125](#), [160](#), [189](#), [349](#), [442](#), [545](#), [547](#)
length-selector (R722), [34](#), [76](#), [76](#), [77](#)
letter, [58](#), [58](#), [59](#), [61](#), [136](#), [167](#), [170](#)
letter-spec (R869), [136](#), [136](#)
level-1-expr (R1003), [167](#), [167](#), [168](#), [171](#)
level-2-expr (R1007), [167](#), [167](#), [168](#), [171](#)
level-3-expr (R1011), [168](#), [168](#), [169](#)
level-4-expr (R1013), [168](#), [169](#)
level-5-expr (R1018), [169](#), [169](#), [170](#)
 lexical token, [19](#), [33](#), [59](#), [62](#)
 LGE, [79](#), [443](#)
 LGT, [79](#), [443](#)
 line, [19](#), [62–65](#)
 linkage association, [5](#), [568](#), [576](#), [579](#), [579](#)
 list-directed formatting, [272](#), [315–319](#)
 list-directed input/output statement, [262](#)
 literal constant, [10](#), [53](#), [147](#), [182](#)
literal-constant (R605), [60](#), [60](#), [166](#)
 LLE, [79](#), [444](#)
 LLT, [79](#), [444](#)
 LOCAL, [211](#), [216](#), [218](#), [574](#), [575](#), [581](#), [590](#)
 local identifier, [571](#), [572](#)
 local procedure pointer, [21](#), [366](#)
 local variable, [32](#), [37](#), [39](#), [52](#), [54](#), [112](#), [114](#), [116](#), [119](#), [126](#), [128](#), [159](#), [161](#), [162](#), [346](#), [366](#), [371](#)
local-defined-operator (R1414), [327](#), [328](#), [328](#)
local-name, [327](#), [328](#)
 LOCAL_INIT, [211](#), [216](#), [574](#), [575](#), [581](#), [588](#), [590](#)
 locality, [216](#), [217](#), [574](#), [575](#), [581](#), [588](#), [590](#)

- locality-spec* (R1130), 211, **211**, 212
- LOCK statement, 231, **239**, 241, 242, 495, 496, 498, 588, 591
- lock variable, **32**, 50, 241, 495, 498, 588, 590
- lock-stat* (R1184), 239, **239**
- lock-stmt* (R1183), 46, **239**
- lock-variable* (R1186), 239, **239**, 240, 241, 496, 591
- LOCK_TYPE, 38, 81, 123, 158, 239, **495**
- LOG, 40, **445**
- LOG10, **445**
- LOG_GAMMA, **445**
- LOGICAL, **445**
- logical intrinsic operation, 177
- logical type, 79
- logical-expr* (R1025), 167, 172, 182, **182**, 197, 211, 213–215, 219, 220, 229, 345, 347
- logical-literal-constant* (R725), 60, **79**, 167, 170
- logical-variable* (R904), 146, **146**, 239–241, 283, 286, 287, 591
- LOGICAL16, **496**
- LOGICAL32, **496**
- LOGICAL64, **496**
- LOGICAL8, **496**
- LOGICAL_KINDS, **496**
- loop-control* (R1123), 211, **211**, 213, 214, 218
- lower-bound* (R816), 118, **118**, 119–121
- lower-bound-expr* (R936), 157, **157**, 193
- lower-bounds-expr* (R937), 157, **157**, 158, 159, 193, 195
- lower-cobound* (R812), 116, **116**
- LZ edit descriptor, 313
- LZP edit descriptor, 313
- LZS edit descriptor, 313
- M**
- m* (R1309), 296, **296**, 301, 307, 308
- main program, **19**, 46, 47, 49, 50, 52
- main-program* (R1401), 43, 47, 138, 325, **325**
- mask-expr* (R1047), 197, **197**, 198–201, 211, 212, 214
- masked array assignment, **20**, **197**, 586
- masked array assignment (WHERE), 197
- masked-elsewhere-stmt* (R1048), 197, **197**, 198, 201
- MASKL, **446**
- MASKR, **446**
- MATMUL, **446**
- MAX, 211, 216, 375, 378, **447**
- MAXEXPONENT, **448**
- MAXLOC, 378, **448**
- MAXVAL, **449**
- MERGE, **450**
- MERGE_BITS, **451**
- MIN, 211, 216, **451**
- MINEXPONENT, **451**
- MINLOC, **452**
- MINVAL, **453**
- MOD, 40, **454**
- mode
 - blank interpretation, 255
 - changeable, 251
 - connection, 251
 - decimal edit, 255
 - delimiter, 255
 - halting, 499, 504, **504**, 508, 510, 525, 531, 532, 569, 597
 - IEEE rounding, 499, 500, **503**, 504, 505
 - input/output rounding, 251, 257, 264, 289, **307**, 313, 314, 532
 - leading zero, 256, 264, 313
 - pad, 256
 - sign, 257, 313
 - underflow, 504, **504**, 507, 512, 527, 535, 597
- model
 - bit, 378
 - extended real, 379
 - integer, 379
 - real, 379
- MODULE, 335, **335**, 336, 363, **363**, **367**
- module, **20**, 46, 47, 52, **325**
- module* (R1404), 43, 138, **326**
- module procedure, **22**, 92, 140, 194, 331–336, 342, 345, 354, 362, 363, 367, 368, 371, 373, 374, 492, 572, 574
- module procedure interface body, 139, **336**
- module reference, **23**, 326
- MODULE statement, 325, **326**
- module subprogram, **27**, 47–49, 136, 139, 360, 576
- module-name*, 326–328, 577
- module-nature* (R1410), 327, **327**, 328
- module-stmt* (R1405), 43, 326, **326**
- module-subprogram* (R1408), 44, 326, **326**, 368
- module-subprogram-part* (R1407), 43, 94, 98, 326, **326**, 329, 639
- MODULO, 40, **454**
- MOLD= specifier, **156**, 607
- MOVE_ALLOC, 160, 232, 376, **454**
- mp-subprogram-stmt* (R1542), 45, 367, **367**
- mult-op* (R1009), 60, 167, 168, **168**

mult-operand (R1005), 167, **167**, 168, 171
multiple-subscript (R920), 151, **151**, 152
multiple-subscript-triplet (R923), 151, **151**, 152
 MVBITS, 376, 377, **456**

N

n (R1316), 297, **297**, 312
 name, **20**, 56, 59, 571
name (R603), 34, 56, 59, **59**, 60, 111, 135, 146, 211, 217, 226, 270, 342, 365
 name association, **5**, 56, **576**, 582
 name-value subsequence, **319**, 320
 NAME= specifier, 111, **114**, 130, 283, 284, **286**, 342, **342**, 364, **568**
 named constant, **10**, 38, 53, 56, 59, 67, 72, 75–77, 103, 105, 121, 124, 125, 129, 131, 132, 134, 141, 147, 370, 572
named-constant (R606), 60, **60**, 65, 74, 75, 102, 134, 577
named-constant-def (R856), 134, **134**, 577
 NAMED= specifier, 283, **286**
 namelist formatting, 272, 319–324
 namelist input/output statement, **262**
 NAMELIST statement, **140**, 206, 320, 328
namelist-group-name, 140, 141, 260–262, 269, 271, 294, 320, 324, 328, 577, 591
namelist-group-object (R873), 140, **140**, 270, 272, 278, 293, 319, 320, 323, 324, 328
namelist-stmt (R872), 45, **140**, 577, 591
 NaN, **16**, 302–306, 310, 388, 423, 427, 470, 474, 478, 501, 505, 508, 509, 513, 532, 533
 NEAREST, **456**
 NEW_INDEX= specifier, **238**, 594
 NEW_LINE, 309, **457**
 NEWUNIT= specifier, 250, 254, **256**, 275, 588, 591, 607
 NEXT, **457**
 NEXTREC= specifier, 283, **287**
 NINT, **458**
 NML= specifier, 260, **262**, 591
 NON_INTRINSIC module nature, **327**
 NON_OVERRIDABLE attribute, **92**, 93
 NON_RECURSIVE attribute, 333, 363, **363**, 367, 369
 nonadvancing input/output statement, 247
 nonblock DO construct, **599**
 NONE, **136**, **138**, **211**
 nonexecutable statement, **25**, 47
nonlabel-do-stmt (R1122), 211, **211**, 212
 NOPASS, **86**, **88**, **93**

NOPASS attribute, *see* PASS attribute
 NORM2, **458**
 normal number, **505**
 normal termination, 16, 49, 50, **50**, 96, 229, 230, 244, 258, 436, 481, 498
 NOT, **458**
not-op (R1019), 60, 169, **169**
 notify variable, **32**, 50, 232, 496
 NOTIFY WAIT statement, 156, **230**, 232, 496, 588, 593
notify-variable (R1167), 155, 230, **230**, 496, 591
notify-wait-stmt (R1166), 46, **230**
 NOTIFY= specifier, **155**, 231, 232, 496, 588
 NOTIFY_TYPE, 81, 123, 158, 230, **496**
 NULL, 101, 111, 183, 186, 355, 377, **459**, 580
null-init (R806), 88, 89, 111, **111**, 113, 131–133, 342, 343
 NULLIFY statement, **161**
nullify-stmt (R942), 46, **161**, 591
 NUM_IMAGES, 185, 186, **460**, 490
 NUMBER= specifier, 283, **287**
 numeric conversion, 190
 numeric editing, 300
 numeric intrinsic operation, 174
 numeric sequence type, **24**, **82**, 142, 144, 145, 584, 587
 numeric storage unit, **26**, 145, 497, 582, 587, 589
 numeric type, **29**, 71–75, 173–175, 178, 181, 190, 418, 447, 464, 482
numeric-expr (R1028), 182, **182**
 NUMERIC_STORAGE_SIZE, **497**

O

O edit descriptor, 307
 object, **11**, 51–54
 object designator, **13**, 52, 53, 127, 131, 148, 184, 319, 320
object-name (R804), 111, **111**, 129, 130, 134, 135, 146, 154, 155, 216, 577
 obsolescent feature, 35, 36, 41, 42, 599–601
octal-constant (R775), 106, **106**
 ONLY, 138, **138**, 139, **327**, 328, 329, 578, 633, 634
only (R1412), 327, **327**, 328
only-use-name (R1413), 327, **327**, 328
 OPEN statement, 41, 244, 245, 250–253, **253**, 258, 262, 271, 272, 276, 286, 290, 307, 322, 588, 591, 594, 595, 625, 627–629
open-stmt (R1204), 46, **253**, 372
 OPENED= specifier, 283, **287**

- operand, **20**
- operation, **66**
 - defined, **12**, **92**, **170**, **179**, **180**, **180**, **181–183**, **211**, **331**, **338**, **344**, **359**, **372**
 - elemental, **14**, **172**, **183**, **184**, **199**
 - intrinsic, **171**, **173–179**
 - logical, **177**
 - numeric, **174**
 - relational, **178**
- OPERATOR, **67**, **93**, **180**, **327**, **335**, **338**, **635**
- operator, **20**, **60**
 - character, **168**
 - defined binary, **169**
 - defined unary, **167**
 - elemental, **14**, **172**, **500**
 - logical, **169**
 - numeric, **167**
 - relational, **168**
- operator precedence, **170**
- OPTIONAL attribute, **69**, **124**, **124**, **128**, **134**, **184**, **205**, **212**, **334**
- optional dummy argument, **356**
- OPTIONAL statement, **133**, **206**
- optional-stmt* (R854), **45**, **133**
- or-op* (R1021), **60**, **169**, **169**
- or-operand* (R1016), **169**, **169**
- other-specification-stmt* (R513), **44**, **45**
- OUT, **122**
- OUT_OF_RANGE, **460**
- output statement, **259**, **309**
- output-item* (R1217), **259**, **260**, **265**, **265**, **278**, **283**
- OUTPUT_UNIT, **250–252**, **256**, **274**, **497**
- override, **89**, **98**, **110**, **111**, **136**, **272**, **300**, **584**
- P**
- P edit descriptor, **313**
- PACK, **461**
- pad mode, **256**
- PAD= specifier, **40**, **41**, **254**, **256**, **260**, **261**, **264**, **276**, **283**, **287**
- padding, **378**, **378**, **437**, **468**
- PARAMETER attribute, **53**, **103**, **112**, **124**, **124**, **125**, **134**, **147**
- PARAMETER statement, **134**, **136**, **330**
- parameter-stmt* (R855), **44**, **134**, **577**
- parent component, **9**, **91**, **95**, **97**, **100**, **584**, **615**
- parent data transfer statement, **264**, **274**, **272–276**, **293**, **318**
- parent team, **28**, **49**, **55**, **156**, **209**, **236**, **238**, **424**, **430**, **436**, **460**, **481**, **484**, **485**, **497**
- parent type, **30**, **81**, **84**, **91**, **95–98**, **340**, **615**
- parent-identifier* (R1418), **329**, **329**
- parent-string* (R909), **117**, **147**, **147**
- parent-submodule-name*, **329**
- parent-type-name*, **80**
- PARENT_TEAM, **430**, **497**
- parentheses, **181**
- PARITY, **462**
- part-name*, **148**, **149**, **155**
- part-ref* (R912), **117**, **131**, **141**, **148**, **148**, **149–151**, **153**, **155**, **413**, **435**, **441**, **485**, **490**
- partially associated, **583**
- PASS attribute, **86**, **88**, **93**, **344**
- passed-object dummy argument, **20**, **88**, **93**, **98**, **340**, **341**, **347**, **647**
- PAUSE statement, **598**
- pending affector, **114**, **263**, **268**, **570**
- PENDING= specifier, **283**, **284**, **287**
- POINTER, **85**, **86**, **87**
- pointer, **20**, **55**, **82**, **86**, **87**, **155**, **161**, **162**, **164**, **186**, **332–334**, **350**, **372**, **481**, **538**, **564**, **579**, **644**
 - procedure, **542**
- pointer assignment, **21**, **120**, **122**, **161**, **191**, **192–194**, **356**, **580**
- pointer assignment statement, **67**, **88**, **101**, **182**, **192**, **195**, **201**, **386**
- pointer association, **5**, **53**, **57**, **96**, **98**, **101**, **117**, **123**, **125**, **127**, **128**, **148**, **162**, **164**, **192**, **194**, **195**, **216**, **217**, **232**, **235**, **268**, **332**, **348**, **350**, **353**, **355**, **356**, **365**, **367**, **382**, **394**, **395**, **455**, **459**, **541–543**, **555**, **566**, **570**, **579–644**
- pointer association context, **122**, **125**, **372**, **591**
- pointer association status, **579**
- POINTER attribute, **67**, **69**, **70**, **79**, **85**, **111**, **120**, **121**, **125**, **125**, **127**, **132**, **134**, **148**, **149**, **152**, **161**, **193**, **194**, **205**, **216**, **223**, **224**, **332**, **334–336**, **339**, **340**, **342**, **346**, **348**, **352**, **354**, **356–358**, **364**, **372**, **374**, **545**, **550**, **566**, **579**, **581**, **585**, **586**, **608**, **646**, **650**
- POINTER statement, **134**, **330**
- pointer-assignment-stmt* (R1034), **46**, **193**, **193**, **200**, **372**, **591**
- pointer-decl* (R858), **134**, **134**
- pointer-object* (R943), **161**, **161**, **591**
- pointer-stmt* (R857), **45**, **134**, **577**
- polymorphic, **21**, **38**, **39**, **70**, **71**, **88**, **101**, **120**, **122**, **148**, **161**, **182**, **188**, **189**, **194**, **195**, **205**, **212**

- 215, 224, 225, 227, 266, 272, 332–334, 344, 348–350, 352, 371, 372, 408, 424, 450, 455, 471, 481, 579, 585
- POPCNT, **462**
- POPPAR, **463**
- POS= specifier, 247, 248, 260, 261, 264, **264**, 283, **288**, 595
- position edit descriptor, 311
- position-edit-desc* (R1315), 296, **297**
- position-spec* (R1227), 280, **280**
- POSITION= specifier, 253, 254, **257**, 283, **288**, 627
- positional arguments, 376
- potential subobject component, **9**, 38, 79–81, 86, 115, 127, 128, 157, 158, 163, 164, 188, 351, 357, 359, 372, 409, 494–496
- power-op* (R1008), 60, 167, 168, **168**
- pre-existing, **584**
- precedence of operators, 170
- PRECISION, 73, **463**, 524
- preconnected, **21**, 245, 251–253, 256, 262, 269, 493, 494, 497
- preconnection, **253**
- prefix* (R1529), 363, **363**, 364, 366
- prefix-spec* (R1530), 363, **363**, 364, 371, 373, 374
- PRESENT, 70, 124, 184, 185, 356, 377, **463**, 645
- present, **356**
- PREVIOUS, **463**
- primary, 166
- primary* (R1001), 166, **166**, 167, 168, 370
- PRINT statement, 245, 251, 255, **259**, 269, 274, 276, 279
- print-stmt* (R1212), 46, **259**, 372
- PRIVATE attribute, 82, 98, 113, **113**, 129, 372, 633
- PRIVATE statement, **91**, **92**, 94, **129**, 328
- private-components-stmt* (R745), 81, 91, **91**
- private-or-sequence* (R729), 80, 81, **81**
- proc-attr-spec* (R1514), 341, **341**, 342
- proc-component-attr-spec* (R742), 86, **86**, 87
- proc-component-def-stmt* (R741), 85, 86, **86**
- proc-component-ref* (R1040), 194, **194**, 344, 345, 357
- proc-decl* (R1515), 86, 89, 341, 342, **342**, 343
- proc-interface* (R1513), 86, 341, **341**, 342
- proc-language-binding-spec* (R1531), 341, 342, 364, **364**, 365, 366, 370, 551
- proc-pointer-init* (R1517), 342, **342**
- proc-pointer-name* (R862), 135, **135**, 161, 194
- proc-pointer-object* (R1039), 193, 194, **194**, 200, 591
- proc-target* (R1041), 99–101, 193, 194, **194**, 200, 356
- PROCEDURE, **86**, **92**, **341**, **367**
- procedure, **21**, 57, 124, 334
- characteristics of, 332
- dummy, **21**, 122, 136, 140, 185, 194, 332, 334–336, 340–342, 345, 354, 355, 361, 362, 364, 371, 373, 374, 569, 572, 578
- elemental, **14**, 54, 183, 194, 342, 345, 356, 361, 363, 373, 374, **374**, 376, 377
- external, **21**, 35, 36, 47, 92, 122, 136, 194, 236, 331, 332, 334, 336, 337, 341, 342, 345, 354, 355, 362, 571, 572, 576, 578, 635, 639, 642
- internal, **21**, 47, 194, 331–334, 345, 346, 354, 362, 364, 367, 394, 569, 572, 574, 577
- intrinsic, 376–491
- module, **22**, 92, 140, 194, 331–336, 342, 345, 354, 362, 363, 367, 368, 371, 373, 374, 492, 572, 574
- non-Fortran, 369
- pure, **22**, 37–39, 98, 196, 200, 211, 332, 334–336, 343, 354, 363, 364, 369, **371**, **372**, 373, **373**, 376, 468
- simple, **22**, 98, 196, 332, 334, 336, 343, 354, 364, 371, **373**, 376, 382, 454, 456, 479, 486, 492, 500, 506, 508, 540
- type-bound, **22**, 79–81, 88, 93, **93**, 94, 95, 97, 98, 191, 278, 327, 338, 344, 347, 349, 362, 371, 373, 374, 572, 574
- procedure declaration statement, 122, 333, 336, **341**, 369, 386, 574
- procedure designator, **13**, 54
- procedure interface, 333
- procedure pointer, **20**, 47, 49, 67, 86–89, 100, 101, 111, 122, 123, 125, 126, 135, 143, 166, 194, 195, 204, 265, 332, 336, 341, 345–347, 354–356, 361, 365, 367, 371, 373, 374, 394, 459, 542, 544, 569, 573, 577, 581, 644
- procedure reference, **23**, 39, 53, 124, 150, 275, 332, 338, 344, 347
- generic, 340
- resolving, 360
- type-bound, 362
- PROCEDURE statement, 334, 335, **337**
- procedure-component-name*, 194
- procedure-declaration-stmt* (R1512), 44, **341**, 342, 343
- procedure-designator* (R1522), 344, **344**, 357, 362
- procedure-entity-name*, 342, 343

- procedure-name*, 92, 93, 194, 196, 335, 342, 344, 345, 367
- procedure-stmt* (R1506), 334, 335, **335**
- processor, **22**, 35, 36, 57
- processor dependent, **22**, 36, 57, 592–597
- procptr-entity-name*, 134
- PRODUCT, **464**
- program, **22**, 35, 36, 47
- program* (R501), **43**
- PROGRAM statement, **325**
- program unit, **22**, 35, 43, 46–50, 56, 58, 59, 61–65, 83, 126, 136, 250, 251, 258, 325, 329, 430, 567, 571, 572, 580, 600, 609, 632–635, 638–640, 642, 655
- program-name*, 325
- program-stmt* (R1402), 43, 325, **325**
- program-unit* (R502), 34, 43, **43**, 47
- PROTECTED attribute, 38, 125, **125**, 126, 135, 142, 216, 327
- PROTECTED statement, **134**
- protected-stmt* (R859), 45, **134**
- PUBLIC attribute, 98, 113, **113**, 129, 633
- PUBLIC statement, **129**, 328
- PURE, **363**, 364, 369, 371
- pure procedure, **22**, 37–39, 98, 196, 200, 211, 332, 334–336, 343, 354, 363, 364, 369, **371**, **372**, 373, **373**, 376, 468
- Q**
- QUIET= specifier, **229**
- R**
- r* (R1306), 295, **295**, 296, 298
- RADIX, 73, **464**, 500, 524
- RANDOM_INIT, **465**, 466, 596
- RANDOM_NUMBER, 466, **466**
- RANDOM_SEED, 377, 465, **466**
- RANGE, 72, 73, **467**, 524
- RANK, 70, 118, 120, **126**, 127, **467**
- rank, **22**, 52–56, 86–88, 94, 95, 100, 101, 110, 115, 117–121, 134, 145, 148, 149, 151, 153, 155, 157, 158, 160, 180, 182, 183, 188–190, 192, 193, 195, 196, 205, 234, 333, 338–340, 348, 350, 352, 356, 361, 374, 386, 391, 392, 413–416, 420, 425, 426, 432, 433, 438, 441, 447–449, 452–455, 458, 459, 461, 462, 464, 466, 468, 469, 475, 478, 480, 482, 485, 488–490, 537, 541, 550, 575, 583, 646–648
- RANK (*), 120, **223**
- RANK DEFAULT, 121, **223**
- rank-clause* (R830), 110, 119, 120, **126**, 127
- rbracket* (R781), 85, 107, **107**, 110, 111, 129, 130, 135, 155, 157
- RC edit descriptor, 314
- RD edit descriptor, 314
- READ (FORMATTED), 272, 273, **335**
- READ (UNFORMATTED), 272, 273, **335**
- READ statement, 41, 53, 246, 251, 255, **259**, 269, 274, 275, 279, 282, 291, 590, 625–627, 629, 631
- read-stmt* (R1210), 46, **259**, 260, 372, 591
- READ= specifier, 283, **288**
- READWRITE= specifier, 284, **288**
- REAL, 149, 190, 378, 379, **467**, 501
- real and complex editing, 301
- real model, 379
- real part, **74**
- real type, 73–74, 74
- real-literal-constant* (R714), 60, 73, **73**
- real-part* (R719), 74, **74**
- REAL128, **497**
- REAL16, **497**
- REAL32, **497**
- REAL64, **497**
- REAL_KINDS, **497**
- REC= specifier, 248, 260, 261, **264**
- RECL= specifier, 254, **257**, 271, 272, 284, **288**, 290, 589, 594
- record, **23**, 243
- record file, **23**, **243**, 245, 247, 249
- record number, **246**
- RECURSIVE, 76, 363, **363**, 369
- recursive input/output statement, **293**
- REDUCE, **211**, 216, 217, **468**
- reduce-operation* (R1131), 211, **211**, 212, 216, 217
- reference, **23**, 54
 - procedure, 39
- rel-op* (R1014), 60, 168, **168**, 178, 502
- relational intrinsic operation, 178
- rename* (R1411), 327, **327**, 328, 572
- rep-char*, 77, **77**, 297, 316, 321
- REPEAT, **469**
- repeat specification, **295**
- representation method, 71, 73, 75, 79
- RESHAPE, 109, **470**
- resolving procedure reference, 360
- resolving procedure references
 - defined input/output, 278
- restricted expression, **184**

- RESULT, 365, **365**, 367, 368
result-name, 365, 368, 577
 RETURN statement, 50, 96, 126, 145, 162, 163, 207, 209, 215, **369**, 541, 590
return-stmt (R1545), 46, 49, 369, **369**
 REWIND statement, 244, 245, 248, 276, 279, 281, **281**, 626
rewind-stmt (R1226), 46, **280**, 372
 RN edit descriptor, 314
 round edit descriptor, 314
round-edit-desc (R1320), 296, **297**
 ROUND= specifier, 254, **257**, 260, 261, **264**, 276, **284**, **289**, 314
 rounding mode
 IEEE, 499, 500, **503**, 504, 505, 511, 523, 526, 533
 input/output, 251, 257, 264, 289, **307**, 313, 314, 532
 RP edit descriptor, 314
 RRSPACING, **470**
 RU edit descriptor, 314
 RZ edit descriptor, 314
- S**
- S edit descriptor, 313
 SAME_TYPE_AS, 185, **470**
 SAVE attribute, 41, 55, 89, 96, 112–115, 126, **126**, 130, 135, 142, 145, 163, 216, 343, 371, 581
 SAVE statement, **135**, 206, 207, 330, 573
save-stmt (R860), 45, **135**, 577
 saved, **23**, 580, 586
saved-entity (R861), 135, **135**, 206
 scalar, **23**, 374
scalar-expr, 107
scalar-xyz (R403), 34, **34**
 SCALE, **471**
 scale factor, **296**, 313
 SCAN, **472**
 scoping unit, **24**, 46, 48, 49, 52, 56, 57, 77, 82, 83, 92, 96, 100, 112–114, 122, 124, 126, 129, 135, 136, 138–141, 143–145, 162, 185, 194, 207, 212, 216, 217, 263, 266, 267, 326–328, 333, 336, 340, 360–363, 365, 368, 370, 499, 501, 567, 572–575, 577, 578, 581, 583, 584, 588, 633, 642, 647
 section subscript, 153
section-subscript (R921), 148, 149, 151, **151**, 153–155
 segment, **24**, 159, 163, 208–210, 231, 232, **232**, 233–236, 238–240, 380, 422, 455, 493, 494, 496
 SELECT CASE construct, 50, **220**, 600, 617
 SELECT CASE statement, 63, **220**
 SELECT RANK construct, 50, 55, 120, 121, 205, **223**, 357, 574, 575, 591
 SELECT RANK statement, 55, 122, **223**, 579
 SELECT TYPE construct, 50, 55, 67, 69, 205, **225**, 357, 574, 575, 579, 591
 SELECT TYPE statement, 55, 63, **225**, 579
select-case-stmt (R1143), 220, **220**, 221, 228
select-construct-name, 223, 225, 226
select-rank-case-stmt (R1152), 223, **223**, 224
select-rank-construct (R1150), 45, 223, **223**
select-rank-stmt (R1151), 223, **223**, 228
select-type-construct (R1154), 45, **225**, 226
select-type-stmt (R1155), 225, **225**, 226, 228
 SELECTED_CHAR_KIND, 75, **472**
 SELECTED_INT_KIND, 72, 84, **473**
 SELECTED_LOGICAL_KIND, **473**
 SELECTED_REAL_KIND, 73, 377, **474**, 609
 selector, **204**
selector (R1105), 204, **204**, 205, 207, 208, 223, 225–227, 357, 579, 591
 separate module procedure, **367**
 separate module subprogram statement, **367**
separate-module-subprogram (R1541), 45, 326, 367, **367**
 sequence, **24**
 sequence association, 355
 SEQUENCE attribute, 80, **81**, 82, 83, 96, 143, 193, 195, 226, 548
 SEQUENCE statement, **81**
 sequence structure, **24**
 sequence type, **24**, 39, 80, 82, **82**, 141, 142, 549, 583
 character, **24**, **82**, 142, 144, 145, 584, 587
 numeric, **24**, **82**, 142, 144, 145, 584, 587
sequence-stmt (R731), 81, **81**
 sequential access, **245**
 sequential access data transfer statement, **264**
 SEQUENTIAL= specifier, 284, **289**
 SET_EXPONENT, **474**
 SHAPE, 70, **475**
 shape, **24**, 54, 232
 SHARED, **211**, 216, 217
 SHIFTA, **475**
 SHIFTL, **475**
 SHIFTR, **476**
 sibling teams, **28**, 49, 238, 436, 460, 484

- SIGN, 40, 41, 73, 476
sign (R712), 72, 72, 73, 302
 sign mode, 257, 300, 313
sign-edit-desc (R1321), 296, 297
 SIGN= specifier, 254, 257, 260, 261, 265, 284, 289, 313
signed-digit-string (R710), 72, 74, 301, 302
signed-int-literal-constant (R707), 72, 72, 74, 131, 296
signed-real-literal-constant (R713), 73, 74, 131
significand (R715), 73, 73
 SIMPLE, 363, 364, 371
 simple procedure, 22, 98, 196, 332, 334, 336, 343, 354, 364, 371, 373, 376, 382, 454, 456, 479, 486, 492, 500, 506, 508, 540
 simply contiguous, 25, 154, 155, 155, 195, 348, 350, 351, 353, 355, 543
 SIN, 476
 SIND, 477
 SINH, 477
 SINPI, 477
 SIZE, 70, 478
 size, 25, 54
 size of a common block, 144
 SIZE= specifier, 260, 265, 284, 289, 291, 292, 587, 590, 626
source-expr (R932), 156, 157, 157, 158, 160, 161, 372, 580, 581
 SOURCE= specifier, 156, 158, 161, 372, 494, 497, 587, 589, 607
 SP edit descriptor, 313
 SPACING, 478
 special character, 59
 specific interface, 18, 278, 335, 336, 337, 342, 360, 361
 specific interface block, 18, 336
 specific name, 25
specific-procedure (R1507), 335, 335, 337
 specification, 110–145
 specification expression, 25, 49, 50, 84, 96, 114, 150, 184, 185, 185, 186, 206, 368, 501, 600, 607
 specification function, 185
 specification inquiry, 185
specification-construct (R508), 44, 44
specification-expr (R1029), 112, 116, 118, 184, 184, 374
specification-part (R504), 43, 44, 44, 45, 49, 92, 113, 114, 128, 185, 187, 325, 326, 329, 330, 334, 335, 364, 366, 367, 371, 373, 374
 SPLIT, 479
 SPREAD, 480
 SQRT, 40, 480, 506, 533, 534
 SS edit descriptor, 313
 standard intrinsic, 18, 35, 492, 604
 standard-conforming program, 35
stat-variable (R946), 50, 155, 156, 158, 162, 164, 164, 165, 228, 230, 231, 233, 234, 237–242, 254, 258, 260, 279, 280, 282, 283, 291, 292, 440, 588, 590, 591, 593
 STAT= specifier, 155, 156, 160, 162, 164, 210, 231, 233, 240, 497, 498, 587, 588, 590, 593, 605
 STAT_FAILED_IMAGE, 50, 156, 164, 241, 242, 380, 381, 425, 455, 497, 498
 STAT_LOCKED, 241, 242, 498, 498
 STAT_LOCKED_OTHER_IMAGE, 241, 242, 498, 498
 STAT_STOPPED_IMAGE, 164, 241, 381, 455, 481, 498, 498
 STAT_UNLOCKED, 241, 242, 498, 498
 STAT_UNLOCKED_FAILED_IMAGE, 241, 242, 498, 498
 statement, 25, 62
 accessibility, 128
 ALLOCATABLE, 129
 ALLOCATE, 67, 69, 76, 77, 117, 120, 156, 160, 164, 165, 194, 231, 494, 497, 559, 560, 580, 581, 587, 589, 593
 arithmetic IF, 599
 ASSIGN, 598
 assigned GO TO, 598
 assignment, 38, 53, 67, 95, 187, 201, 231, 232, 496, 536, 586, 588
 ASSOCIATE, 55, 204, 579
 ASYNCHRONOUS, 130, 206, 330, 574, 577
 attribute specification, 128–145
 BACKSPACE, 244, 248, 276, 279, 280, 281, 626, 627
 BIND, 130, 330, 567, 573
 BLOCK, 112, 116, 119, 206, 588
 BLOCK DATA, 62, 325, 330
 CALL, 228, 232, 331, 344, 359, 360, 369, 455
 CASE, 220
 CHANGE TEAM, 49, 55, 207, 231, 241, 380, 579
 CLASS DEFAULT, 225
 CLASS IS, 225, 424
 CLOSE, 244, 245, 250, 252, 253, 258, 258, 276, 279, 626
 COMMON, 141, 143, 143–145, 206, 328, 330, 573, 582–584, 599

component definition, **69, 85**
computed GO TO, **228, 229, 599, 600**
CONTAINS, **48, 92, 369**
CONTIGUOUS, **130**
CONTINUE, **229, 598**
CRITICAL, **181, 209, 231, 232, 242**
CYCLE, **204, 210, 214, 215, 601**
DATA, **37, 38, 41, 48, 107, 112, 130, 145, 206, 330, 459, 574, 575, 577, 586, 599, 600**
data transfer, **41, 61, 243–246, 248–252, 259, 265, 268–270, 275, 278, 279, 281, 290–295, 307, 313, 315, 317, 318, 320, 322, 495, 586, 587, 589, 595, 626, 629, 630**
DEALLOCATE, **162, 164, 165, 231, 494, 497, 560, 593**
defined assignment, **39, 192, 192, 359, 591**
derived type definition, *see* statement, TYPE
DIMENSION, **133, 330**
DO, **210, 586, 599, 601**
DO CONCURRENT, **69, 200, 210**
DO WHILE, **210**
ELSE, **219**
ELSE IF, **63, 219**
ELSEWHERE, **63, 197**
END, **14, 49, 126, 145, 162, 163, 232, 541, 590**
END ASSOCIATE, **63, 204**
END BLOCK, **63, 163, 206**
END BLOCK DATA, **63, 330**
END CRITICAL, **63, 181, 209, 231, 232**
END DO, **63, 212**
END ENUM, **63, 102**
END ENUMERATION TYPE, **105**
END FORALL, **62, 200**
END FUNCTION, **63, 365**
END IF, **63, 219, 598**
END INTERFACE, **63, 334**
END MODULE, **63, 326**
END PROCEDURE, **63, 367**
END PROGRAM, **63, 325**
END SELECT, **63, 220, 226**
END SUBMODULE, **63, 329**
END SUBROUTINE, **62, 366**
END TEAM, **49, 62, 207, 228, 231, 241, 380**
END TYPE, **63, 81**
END WHERE, **63, 197**
ENDFILE, **62, 244, 245, 248, 255, 276, 279, 281, 626**
ENTRY, **48, 141, 179, 180, 192, 326, 331, 336, 363, 365, 367, 374, 573, 582–584, 599, 601**
ENUM, **102**
ENUMERATION TYPE, **105**
ENUMERATOR, **102**
EQUIVALENCE, **141, 141, 141–145, 206, 328, 330, 582–584, 599, 601**
ERROR STOP, **50, 51, 229, 593**
EVENT POST, **231, 237, 237, 238, 241, 422, 493, 494, 588, 591, 594**
EVENT WAIT, **231, 237, 237, 241, 493, 494, 588, 591, 594**
executable, **25, 47**
EXIT, **204, 215, 228**
EXTERNAL, **122, 341**
FAIL IMAGE, **230**
file inquiry, **282**
file positioning, **244, 280**
FINAL, **94**
FLUSH, **245, 279, 282, 292**
FORALL, **69, 172, 201, 574, 575, 586**
FORM TEAM, **49, 208, 231, 238, 241, 588, 590, 591, 594**
FORMAT, **35, 48, 61, 206, 262, 294, 294, 326**
formatted input/output, **243, 261**
FUNCTION, **69, 70, 136, 179, 180, 184, 325, 363, 364, 367–369, 573**
GENERIC, **92, 93, 337, 337, 340, 360**
GO TO, **63, 228, 228**
IF, **171, 220**
IMPLICIT, **136, 141, 330**
IMPLICIT NONE, **136**
IMPORT, **48, 138, 206, 571, 574, 578**
input, **259, 260, 626**
input/output, **243–293, 586**
INQUIRE, **40, 245, 247, 249, 250, 252, 264, 275, 276, 279, 282, 292, 293, 495, 587, 589, 591, 595, 625**
INTENT, **133, 206**
INTERFACE, **334, 635**
INTRINSIC, **330, 343**
intrinsic assignment, **39, 100, 155, 163, 165, 182, 188, 193, 194, 216, 242, 250, 282, 293, 322, 372, 377, 381, 388, 408, 486, 586, 593**
list-directed input/output, **262**
LOCK, **231, 239, 241, 242, 495, 496, 498, 588, 591**
MODULE, **325, 326**
NAMELIST, **140, 206, 320, 328**
namelist input/output, **262**
nonexecutable, **25, 47**

NOTIFY WAIT, 156, **230**, 232, 496, 588, 593
 NULLIFY, **161**
 OPEN, 41, 244, 245, 250–253, **253**, 258, 262, 271, 272, 276, 286, 290, 307, 322, 588, 591, 594, 595, 625, 627–629
 OPTIONAL, **133**, 206
 output, **259**, 309
 PARAMETER, **134**, 136, 330
 PAUSE, **598**
 POINTER, **134**, 330
 pointer assignment, 67, 88, 101, 183, **192**, 195, 201, 386
 PRINT, 245, 251, 255, **259**, 269, 274, 276, 279
 PRIVATE, **91**, **92**, 94, **129**, 328
 PROCEDURE, 334, 335, **337**
 procedure declaration, 122, 333, 336, **341**, 369, 386, 574
 PROGRAM, **325**
 PROTECTED, **134**
 PUBLIC, **129**, 328
 READ, 41, 53, 246, 251, 255, **259**, 269, 274, 275, 279, 282, 291, 590, 625–627, 629, 631
 RETURN, 50, 96, 126, 145, 162, 163, 207, 209, 215, **369**, 541, 590
 REWIND, 244, 245, 248, 276, 279, 281, **281**, 626
 SAVE, **135**, 206, 207, 330, 573
 SELECT CASE, 63, **220**
 SELECT RANK, 55, 122, **223**, 579
 SELECT TYPE, 55, 63, **225**, 579
 separate module subprogram, **367**
 SEQUENCE, **81**
 statement function, 48, 76, 206, 326, 360, 368, **370**, 574, 575, 600
 STOP, 50, **229**, 232, 373, 593
 SUBMODULE, 325, **329**
 SUBROUTINE, 192, 325, 363, **366**, 368, 369
 SYNC ALL, 209, 231, **233**, 234, 235, 241
 SYNC IMAGES, 231, **234**, 241, 242
 SYNC MEMORY, 231, **235**, 241, 651
 SYNC TEAM, 209, 231, **236**, 241
 TARGET, **135**, 330
 TYPE, **80**, 83, 84, 113, 577
 type declaration, 69, 70, 89, 110, **110**, 110–113, 122, 136, 141, 145, 185, 326, 330, 365, 368, 370, 586
 type guard, 76, 77, **225**
 TYPE IS, **225**, 471
 type parameter definition, **83**
 type-bound procedure, **92**, **93**
 unformatted input/output, 244, **261**
 UNLOCK, 231, **239**, 241, 242, 495, 496, 498, 588, 591
 USE, 48, 83, 129, **326**, 330, 360–362, 572, 574, 576, 578, 632–634, 638
 VALUE, **135**, 206
 VOLATILE, **135**, 206, 330, 574, 577
 WAIT, 252, 264, 279, **279**, 630
 WHERE, 172, **197**
 WRITE, 37–39, 245, 251, 255, **259**, 269, 274, 276, 279, 293, 586, 625, 626, 628, 629
 statement entity, **25**, 216, 571, 572, 574
 statement function, 370, 600
 statement function statement, 48, 76, 206, 326, 360, 368, **370**, 574, 575, 600
 statement keyword, **19**, 56
 statement label, **26**, 61, **61**, 62, 64, 65, 345, 571
 statement order, 48
 STATUS= specifier, 253, 254, 256, **257**, 258, **259**, 594, 595, 628
stmt-function-stmt (R1547), 44, 326, 329, 335, **370**, 577
 STOP statement, 50, **229**, 232, 373, 593
stop-code (R1164), 229, **229**, 230
stop-stmt (R1162), 46, 96, **229**
 stopped image, **16**, 50, 159, 163, 164, 381, 455
 STOPPED_IMAGES, **480**
 storage association, **5**, 56, 57, 141–145, 368, 372, 481, 582–584
 storage sequence, **26**, 80, 82, 142–145, 330, 394, 395, 540–542, 561, 582, **582**, 583
 storage unit, **26**, 141–145, 263, 268, 276, 279, 330, 356, 395, 582–584
 character, **26**, 121, 142, 145, 492, 582, 583, 587, 589
 file, **15**, 243, 244, 246–249, 257, 263, 264, 270, 271, 281, 288–290, 494, 582
 numeric, **26**, 145, 497, 582, 587, 589
 unspecified, **26**, 582, 583, 587, 589
 STORAGE_SIZE, 107, **481**
 stream access, **246**
 stream access data transfer statement, **264**
 stream file, **26**, **243**, 246, 247, 249, 290
 STREAM= specifier, 284, **289**
stride (R924), 151, **151**, 152, 154, 268
 structure, **26**, 51, 52, 80
 structure component, **26**, 132, 148–150, 548, 615
 structure constructor, **26**, 51, 56, 66, 91, 99–101, 132, 133, 182, 184, 186, 459, 495, 573, 611

structure-component (R913), 131, 146, 147, **149**, 155, 157, 161

structure-constructor (R756), **99**, 100, 131, 132, 166, 167, 372

subcomponent, **9**, 89, 99, 194, 580–582, 586–589

submodule, **27**, 46, 47, 52, 139, **329**, 576

submodule (R1416), 43, **329**

submodule identifier, **329**

SUBMODULE statement, 325, **329**

submodule-name, 329, 330

submodule-stmt (R1417), 43, 329, **329**, 330

subobject, **27**, 52, 53, 55, 122, **148**, 349, 580

subprogram, **27**, 46–50, 52, 136

- elemental, **14**, 363, 364, 374
- external, **27**, 46, 331
- internal, **27**, 47–49, 139, 331, 576
- module, **27**, 47–49, 139, 576

subroutine, **27**

- atomic, **27**, 50, 232, 233, 376, 380, 382, 398–402, 422, 492, 497, 590
- collective, **27**, 376, 380–382, 408–411, 425, 481, 497, 498, 590

subroutine reference, 359

SUBROUTINE statement, 192, 325, 363, **366**, 368, 369

subroutine-name, 335, 336, 366, 573

subroutine-stmt (R1538), 43, 335, 336, 363, 364, 366, **366**, 573, 577

subroutine-subprogram (R1537), 27, 43–45, 326, **366**, 367

subroutines

- intrinsic, 376

subscript, 150

- section, 153

subscript (R919), 131, 148, 151, **151**, 152, 154, 268

subscript triplet, 153

subscript-triplet (R922), 151, **151**, 154

substring, **147**

substring (R908), 141, 142, 146, **147**

substring ending point, **147**

substring starting point, **147**

substring-range (R910), 117, 147, **147**, 149, 151, 155, 268

suffix (R1535), 364, **365**, 367

SUM, **482**

SYNC ALL statement, 209, 231, **233**, 234, 235, 241

SYNC IMAGES statement, 231, **234**, 241, 242

SYNC MEMORY statement, 231, **235**, 241, 651

SYNC TEAM statement, 209, 231, **236**, 241

sync-all-stmt (R1168), 46, **233**

sync-images-stmt (R1170), 46, **234**

sync-memory-stmt (R1172), 46, **235**

sync-stat (R1169), 207, 209, 233, **233**, 234–242

sync-team-stmt (R1173), 46, **236**

synchronous input/output, 255, 262, 265, 267

SYSTEM_CLOCK, 37–39, **482**

T

T edit descriptor, 312

TAN, **483**

TAND, **483**

TANH, **484**

TANPI, **484**

target, **28**, 53, 55, 70, 88–90, 95, 101, 113, 115, 117, 120, 122–126, 132, 146, 149, 156, 160–162, 164, 182, 189, 192, 194, 195, 200, 201, 265, 266, 270, 272, 343, 346, 348, 351, 353, 355, 394, 540, 542, 545, 579–582, 585, 587, 590, 591

TARGET attribute, 5, 38, 89, 125, 127, **127**, 135, 141, 145, 161, 162, 193, 205, 216, 224, 332, 334, 339, 349, 350, 352, 353, 357, 358, 410, 455, 468, 540, 543, 545, 566, 567, 580, 581, 590, 608, 644, 645

TARGET statement, **135**, 330

target-decl (R864), 135, **135**

target-stmt (R863), 45, **135**, 577

team, **28**, 49, **49**, 54, 55, 156, 163, 208, 209, 233, 234, 236, 238, 241, 376, 380, 436, 460, 481, 485

- current, 380

team number, **28**, 49, 156, 238, 436

team variable, **32**, 208, 498, 588, 590

team-construct-name, 207

team-number (R1180), 238, **238**

team-value (R1115), 155, 156, 207, **207**, 208, 236

team-variable (R1181), 238, **238**, 240, 241, 591

TEAM= specifier, **155**, 156

TEAM_NUMBER, 185, 186, **484**

TEAM_NUMBER= specifier, **155**, 156

TEAM_TYPE, 85, 115, 148, 157, 158, 191, 208, 238, 424, 430, 436, 460, 481, 484, 485, **498**, 590

THEN, **219**

THIS_IMAGE, 185, 186, **485**

TINY, 478, **486**

TKR compatible, **340**

TL edit descriptor, 312

TOKENIZE, **486**

- totally associated, **583**
- TR edit descriptor, **312**
- TRAILZ, **487**
- TRANSFER, **186, 487**
- transfer of control, **203, 228, 291, 292**
- transformational function, **28, 186, 370, 376, 376, 377, 382, 403, 404, 492, 493, 506**
- TRANSPPOSE, **488**
- TRIM, **488**
- truncation, **378, 437, 468**
- TYPE, **68**
- type, **28, 51, 66–109**
 - abstract, **29, 68, 69, 93, 96, 97, 99, 148, 157**
 - character, **75–79**
 - complex, **74–75**
 - declared, **29, 69, 70, 88, 100, 101, 108, 110, 148, 150, 158, 161, 180, 182, 188, 191, 192, 194, 195, 205, 224, 226, 227, 278, 339, 344, 345, 347, 348, 352, 362, 370, 424, 450, 455, 471, 494, 495, 575**
 - derived, **29, 51, 52, 66, 79–102, 108, 548, 549**
 - dynamic, **29, 70, 95, 97, 98, 101, 108, 128, 158, 159, 161, 164, 180, 182, 189, 191, 192, 194, 195, 205, 225, 226, 232, 235, 278, 344, 352, 362, 424, 450, 455, 470, 471, 481, 575, 579, 585, 616, 659**
 - expression, **182**
 - extended, **29, 84, 91, 95–97, 584, 609, 613**
 - extensible, **29, 68, 80, 81, 88, 96, 272, 424, 471, 615, 649**
 - extension, **29, 70, 96–98, 226, 352, 424, 649**
 - integer, **71–72**
 - intrinsic, **29, 51, 66, 71–79**
 - logical, **79**
 - numeric, **29, 71–75, 173–175, 178, 181, 190, 418, 447, 464, 482**
 - operation, **183**
 - parent, **30, 81, 84, 91, 95–98, 340, 615**
 - primary, **182**
 - real, **73–74, 74**
- type compatible, **30, 70, 71, 88, 157, 158, 188, 193, 340, 349, 394, 455**
- type conformance, **188**
- type declaration statement, **69, 70, 89, 110, 110, 110–113, 122, 136, 141, 145, 185, 326, 330, 365, 367, 370, 586**
- type equality, **82**
- type guard statement, **76, 77, 225**
- TYPE IS statement, **225, 471**
- type parameter, **30, 39, 51, 67, 69–71, 79, 82, 84, 88, 100, 108, 110, 112, 134, 145, 182, 185, 188, 205, 208, 224, 232, 332, 349, 374, 450, 455, 487, 541, 548, 573, 575**
- type parameter definition statement, **83**
- type parameter inquiry, **30, 150, 182, 185**
- type parameter keyword, **19, 56, 99**
- type parameter order, **31, 84**
- type specifier, **68**
 - CHARACTER, **75**
 - CLASS, **70**
 - COMPLEX, **74**
 - derived type, **69**
 - DOUBLE PRECISION, **73**
 - INTEGER, **72**
 - LOGICAL, **79**
 - REAL, **73**
 - TYPE, **69**
- TYPE statement, **80, 83, 84, 113, 577**
- type-attr-spec* (R728), **80, 80, 96**
- type-bound procedure, **22, 79–81, 88, 93, 93, 94, 95, 97, 98, 191, 278, 327, 338, 344, 347, 349, 362, 371, 373, 374, 572, 574**
- type-bound procedure statement, **92, 93**
- type-bound-generic-stmt* (R751), **92, 92, 338**
- type-bound-proc-binding* (R748), **92, 92**
- type-bound-proc-decl* (R750), **92, 92**
- type-bound-procedure-part* (R746), **80, 82, 92, 94, 549**
- type-bound-procedure-stmt* (R749), **92, 92**
- type-declaration-stmt* (R801), **44, 76, 110, 110, 371, 577**
- type-guard-stmt* (R1156), **225, 225, 226**
- type-name*, **80, 81, 83, 93, 99**
- type-param-attr-spec* (R734), **83, 84, 84**
- type-param-decl* (R733), **83, 83, 84**
- type-param-def-stmt* (R732), **80, 83, 83, 84**
- type-param-inquiry* (R916), **150, 150, 166, 167, 573**
- type-param-name*, **80, 83, 84, 86, 150, 166, 167, 573, 577**
- type-param-name-list*, **84**
- type-param-spec* (R755), **56, 99, 99**
- type-param-value* (R701), **67, 67, 68, 76, 77, 86, 99, 111, 157, 159, 363, 601**
- type-spec* (R702), **68, 68, 69, 76, 77, 107, 108, 156–160, 225, 226**
- TYPEOF, **68**

U

UBOUND, 70, 224, **489**
 UCOBOUND, **489**
 ultimate argument, **31**, 163, 195, 348, 349, 354, 355, 357, 381
 ultimate component, **9**, 38, 39, 79, 80, 117, 120, 122, 141, 143, 161, 186, 188, 212, 272, 345, 349, 371, 410, 583
 ultimate entity, **328**
 undefined, **31**, 53, 162, 579–581, 585, 586
 undefinition of variables, 585
 underflow mode, 504, **504**, 507, 512, 527, 535, 597
underscore (R602), 58, **58**
 UNFORMATTED, 272, 273, **335**
 unformatted data transfer, 270
 unformatted input/output statement, 244, **261**
 unformatted record, **244**
 UNFORMATTED= specifier, 284, **290**
 Unicode file, **255**
 unit, **31**, 244–246, 250, **250**, 251–253, 255, 256, 258, 259, 264, 267–270, 274, 279, 280, 282–290, 292, 293, 311, 318, 493, 494, 497, 586, 589, 625, 627–630
 UNIT= specifier, 253, 258, 260, 279, 280, 282, 283
 unlimited polymorphic, **31**, 69, **70**, 108, 143, 157, 158, 193, 226, 352, 394, 424, 471, 481, 659
unlimited-format-item (R1305), 294, 295, **295**, 299
 UNLOCK statement, 231, **239**, 241, 242, 495, 496, 498, 588, 591
unlock-stmt (R1185), 46, **239**
 unordered segments, **232**, 233, 380, 386, 422
 UNPACK, **490**
 unsaved, **31**, 161, 162, 366, 580, 581, 587, 588, 590
 unspecified storage unit, **26**, 582, 583, 587, 589
until-spec (R1178), 237, **237**
 UNTIL_COUNT= specifier, **237**, 422
upper-bound (R817), 118, **118**, 119
upper-bound-expr (R938), 157, **157**, 193
upper-bounds-expr (R939), 157, **157**, 158–160, 193, 195
upper-cobound (R813), 116, **116**
 use association, **5**, 21, 38, 47, 57, 70, 77, 97, 113, 114, 125, 128, 129, 136, 140, 142, 143, 184, 185, 194, **326**, 325–329, 336, 367, 372, 374, 574, 576, 580
 use path, 328
 USE statement, 48, 83, 129, **326**, 330, 360–362, 572, 574, 576, 578, 632–634, 638
use-defined-operator (R1415), 327, 328, **328**

use-name, 327, 328, 572

use-stmt (R1409), 44, 206, **327**, 328, 577

V

v (R1312), 275, 296, **296**, 311
 VALUE attribute, 70, 88, 94, 121, 127, **127**, 128, 135, 216, 268, 332, 334, 335, 338, 339, 348–351, 353, 364, 371, 374, 410, 468, 551, 552, 570, 581, 655, 656
 value separator, **315**
 VALUE statement, **135**, 206
value-stmt (R865), 45, **135**
 variable, **31**, 52–54, 56, 59, 124
 definition & undefinition, 585
variable (R902), 100, 107, 131, 146, **146**, 155, 187, 189–191, 193, 194, 198–201, 204, 216, 217, 225, 230, 237–239, 265, 345, 347, 357, 536, 591
 variable definition, **12**
variable-name (R903), 140, 141, 143, 144, 146, **146**, 147, 157, 161, 193, 211, 212, 577, 591
 vector subscript, **32**, 55, 89, 117, 148, 153, 154, 204, 225, 249, 250, 319, 350, 351, 357, 579, 581, 644
vector-subscript (R925), 151, **151**, 153
 VERIFY, **491**
 VOLATILE attribute, 37, 39, 127, 128, **128**, 135, 193, 195, 205, 212, 216, 327, 328, 332, 334, 351, 353, 371, 576, 577, 582, 588, 590, 614
 VOLATILE statement, **135**, 206, 330, 574, 577
volatile-stmt (R866), 45, **135**

W

w (R1308), 296, **296**, 300, 301, 303–311, 316, 318, 321
 wait operation, 253, 258, 265, 267–269, 278–279, **279**, 280, 282, 287, 288, 291, 292
 WAIT statement, 252, 264, 279, **279**, 630
wait-spec (R1223), 279, **279**
wait-stmt (R1222), 46, **279**, 372
 WHERE construct, **197**
 WHERE statement, 172, **197**
where-assignment-stmt (R1046), 172, 197, **197**, 198, 199, 201
where-body-construct (R1045), 197, **197**, 198
where-construct (R1043), 45, 197, **197**, 199, 201
where-construct-name, 197
where-construct-stmt (R1044), 197, **197**, 198, 201, 228

where-stmt (R1042), 46, 197, **197**, 199, 201
WHILE, **211**, 213, 214
whole array, **32**, 150, **150**, 152, 441, 489
WRITE (FORMATTED), 272, 273, **335**
WRITE (UNFORMATTED), 272, 273, **335**
WRITE statement, 37–39, 245, 251, 255, **259**, 269,
274, 276, 279, 293, 586, 625, 626, 628, 629
write-stmt (R1211), 46, **259**, 260, 372, 591
WRITE= specifier, 284, **290**

X

X edit descriptor, 312
xyz, 34
xyz-list (R401), **34**
xyz-name (R402), **34**

Z

Z edit descriptor, 307
zero-size array, 54, 119, 132